A Topology-Aware Scheduling Strategy for Distributed Stream Computing System

Bo Li[1], Dawei Sun[1(✉)], Vinh Loi Chau[2], and Rajkumar Buyya[3]

[1] School of Information Engineering, China University of Geosciences, Beijing 100083, People's Republic of China
{libocn, sundaweicn}@cugb.edu.cn
[2] School of Information Technology, Deakin University, Geelong, VIC 3216, Australia
vlchau@deakin.edu.au
[3] Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Parkville, Australia
rbuyya@unimelb.edu.au

Abstract. Reducing latency has become the focus of task scheduling research in distributed big data stream computing systems. Currently, most task schedulers in big data stream computing systems mainly focus on tasks assignment and implicitly ignore task topology which can have significant impact on the latency and energy efficiency. This paper proposes a topology-aware scheduling strategy to reduce the processing latency of stream processing systems. We construct the data stream graph as a directed acyclic graph and then, divide it using the graph Laplace algorithm. On the divided graph, tasks will be assigned with a low-latency scheduling strategy. We also provide a computing node selection strategy, which enables the system to run tasks on the topology with the least number of computing nodes. Based on this scheduling strategy, the tasks of the data stream graph can be redistributed and the scheduling mechanism can be optimized to minimize the system latency. The experimental results demonstrate the efficiency and effectiveness of the proposed strategy.

Keywords: Stream computing · Big data system · Topology-aware · Scheduling · Graph division

1 Introduction

With the increase in demand for real-time data processing especially in streaming applications, timeliness of data has become prominent with the rising number of applications deployed in streaming computing platforms across various fields such as finance and banking [1, 2], intelligent recommendation system [3], and Internet of Things [4]. Currently, many big data streaming solutions have been provided [1, 5], such as Storm, Flink, Spark Streaming, etc. Storm is one of the most popular open source big data stream computing systems [7]. It provides powerful distributed cluster management, millisecond latency, rich APIs and high fault tolerance mechanisms, and is widely used in the field of real-time data processing [8].

W. Xiang et al. (Eds.): BROADNETS 2021, LNICST 413, pp. 132–147, 2022.
https://doi.org/10.1007/978-3-030-93479-8_8

Storm's stream processing can be viewed as a directed acyclic graph (DAG) topology [5]. Stream is an abstraction of data transmission between different vertices. It is an unbounded sequence of tuples in time. Storm has two types of vertices: Spout and Bolt. Spout is the source representing the Stream and is responsible for emitting Streams from a specific data source of the topology. Bolt can receive any number of streams as input and then process the data. Bolt can also emit new streams to the downstream Bolt for processing.

System latency and system throughput are important metrics to measure the performance of a stream computing system [7–9]. Therefore, reducing system latency and improving system throughput are the major challenges for task schedulers. Task scheduling for stream computing systems is an effective way to achieve these goals. If tasks are assigned based on the transfer rate between them and the computing resources on the compute nodes, system latency and system throughput can be significantly improved.

The topology-aware scheduling policy is able to place tasks on the appropriate compute nodes based on the structure of the topology. To achieve this goal, we first need to construct DAGs and divide them rationally, and assign tasks to as few compute nodes as possible.

1.1 Contributions

In this paper, our primary focus is to reduce the time delay of distributed stream processing systems. Our contributions are as follows:

(1) We established the DAG model, and a graph partitioning method based on graph Laplacian which can be used to create high-quality partitioning results quickly and efficiently.
(2) Based on the partitioning results, tasks are allocated using a low-latency scheduling strategy. We then proposed a computing node selection strategy to run tasks on the topology with the least number of nodes. We named this approach Ts-Stream.
(3) We conducted experiments to evaluate system performance using time delay as a metric. The experiments demonstrated the effectiveness of our proposed strategy.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 will explain DAG model and communication model. In Sect. 3, the graph partitioning method based on graph Laplacian and the Ts-Stream scheduling strategy are introduced. Section 4 dictates the experiment set up and report the evaluation results. Section 5, summarizes related studies on the scheduling problem. Finally, conclusions and future works are presented in Sect. 6.

2 Related Work

Computational models of Big Data can be divided into batch computing and streaming computing [10]. Batch computing are suitable for different big data application scenarios when data is first stored for computation and the real-time requirements are not a priority. Streaming computing is more suitable for the application scenarios that have strict real-time requirements and do not need to store data first [11].

At the time this work is conducted, researches related to large data batch processing and calculation is relatively mature, forming an efficient and stable batch computing system represented by Google's MapReduce programming model and the opensource Hadoop computing system [6, 12].

In streaming computing, it is impossible to determine the moment of arrival and the order of arrival of data, and it is also impossible to store all the data [13]. Many solutions have been proposed to solve this problem. Yahoo launched S4 Streaming Processing Computing System in 2010 [14], Twitter launched Storm Streaming Computing System in 2011 [5], and Flink originated from a research project called Stratosphere [4]. Storm, S4 and Flink have typical streaming data computing architecture, where data is computed in task topology and outputs valuable information, which has largely driven the development and application of big data streaming computing technology.

Scheduling problem of streaming applications is an active research area [16].

In [17], a Storm-based resource-aware scheduling policy, R-Storm, was proposed. R-Storm considers resource constraints in three main aspects: CPU, memory, and bandwidth. It focuses on memory constraints and considers CPU and network bandwidth constraints as soft constraints.

In [18], a dynamic resource scheduling DRS is proposed to meet the estimation of necessary resources required for real-time demand, effective and efficient re-resource provisioning and scheduling, and effective implementation of such scheduler in cloud-based DSMS.

In [19], a stream computing system T-Storm was proposed. T-Storm monitors data traffic and CPU load changes of each worker node in real time through an external plug-in to achieve fine-grained control and optimized resource usage of the worker nodes.

Thread-level task migration is also an important research direction for task scheduling. In [20], a thread-level task migration N-Storm is proposed, which can perform thread-level task migration without stopping the Storm, avoiding the time wastage due to unnecessary Executor and worker stops and restarts.

The detailed division of the topology is also an important direction to reduce the system delay. In [21], an adaptive hierarchical scheduling P-Scheduler was proposed, which reduces the system latency by dividing the topological graph using the open source graph partitioning software METIS and then dividing the tasks with tightly transmitted data among the same compute nodes after two levels of scheduling.

Ensuring that the system latency is in a reasonable range is also an important direction. In [22], the task assignment problem with delay guarantee (TAPLG) is introduced and two heuristic algorithms, AHA and PHA, are proposed, while considering the critical path of the topology to reduce the latency of the stream topology and reduce the energy consumption.

The above task scheduling works on Storm rarely considers the structure of the topology map. In this work, we proposed Ts-Stream strategy uses the graph Laplacian algorithm to quickly divide the task graph, and uses fewer computing nodes to run the topology, reducing the system latency of the system. The summary of the comparison between our work and other closely related works is given in Table 1.

Table 1. Comparison of Ts-Stream and related work

Parameter	Related work					Ts-Stream
	[17]	[18]	[19]	[20]	[21]	
Task scheduling	✓	✓	✓	✓	✓	✓
Communication saving	✓	✓	✓	✗	✗	✓
Latency saving	✓	✗	✓	✓	✓	✓
Topology aware	✗	✗	✗	✗	✗	✓
Graph partition	✗	✗	✗	✗	✓	✓

3 Problem Statement

This section introduces DAG model and communication model in big data stream computing environments.

3.1 DAG Model

The big data stream processing process can be represented by DAG, in which a vertex represents a Spout or a Bolt, and a directed edge between two vertices forms a Stream between them. Stream is an infinite sequence of tuples and can be considered as an abstraction of data communication between components (Spout or Bolt). A DAG can be represented as $G = (V, E)$,where $V = \{v_1, v_2, \ldots, v_n\}$ represents a finite set of n vertices and $E = \{e_{1,2}, e_{1,3}, \ldots, e_{n-i,n}\}$, $i \in \{1, 2, \ldots, n\}$ is a finite set of directed edges. The Spout component acts as a data source to send tuples to the topology, while Bolt implements the processing of data by the topology and passes the results to the downstream components. Each component can execute multiple tasks to increase the parallelism of the topology.

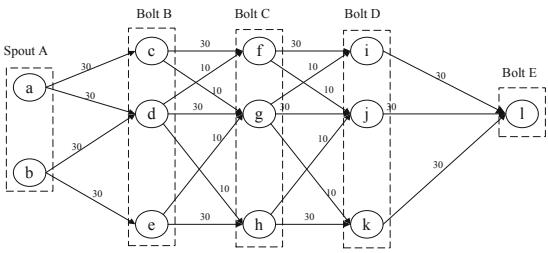

Fig. 1. A topology of storm

Figure 1 shows a topology with five components including one Spout and four Bolts. The number of vertices in each component is also the number of instances of the component.

We define r_{ij} as the transfer rate of tuples between two adjacent vertices, which is the number of tuples sent from v_i to v_j per unit time. Since the arrival rate of the data stream changes, when the data stream fluctuates greatly at a certain moment, if this value is taken at this point in time, the accuracy of the entire DAG will be affected. In order to avoid the effect of violent fluctuations in the data stream at a specific point in time, we take the mathematical expectation E_r of all r_{ij} in the statistical time as the data transfer rate between two vertices.

$$E_r = \frac{1}{n}\sum_{i=1}^{n} r_{ij}, \tag{1}$$

In term of resources required for a task, we only consider CPU to focus more on the scheduling issue. However, the proposed solution can be applied for other types of resources including memory and bandwidth. We denote R_{C_v} as the total amount of resources to be consumed by the whole topology and it can be represented by (2).

$$R_{C_v} = \sum_{i=1}^{n} R_{C_{v_i}}, \tag{2}$$

where $R_{C_{v_i}}$ is the CPU resources consumed by a task.

3.2 Communication Model

In a stream computing system, there are three types of communications causing the overhead problem: processes between compute nodes, processes within a compute node and threads within a process.

Processes between compute nodes usually have higher communication overhead than processes within a compute node. These first two type of communication overheads are more significant than the third type which happens between threads in a process [17]. Therefore, in this research, we will ignore the inter-threads communication and focus on solving the top two types.

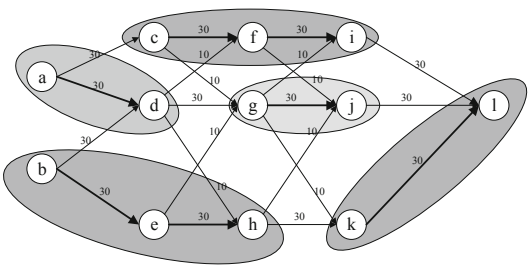

Fig. 2. Task assignment

As shown in Fig. 2, placing adjacent nodes with high communication rates in the same process or computing node can effectively reduce communication delays. $P = \{p_1, p_2, \ldots, p_u\}$ is used to denote the compute nodes in the cluster. The amount of data transfer between compute nodes in the cluster dtv_P is denoted by (3).

$$dtv_P = \sum_{i=1}^{u} \sum_{j=1}^{u} dtv(p_i, p_j), i < j, \tag{3}$$

subjected to

$$dtv(p_i, p_j) = \begin{cases} 0 & \text{no data transmission between } p_i \text{ and } p_j, \\ E_r \cdot q & \text{otherwise,} \end{cases} \tag{4}$$

where $dtv(p_i, p_j)$ denotes the data transfer rate between p_i and p_j, and q denotes the number of task pairs communicated between p_i and p_j.

Use $W = \{w_1, w_2, \ldots, w_h\}$ to denote the processes within the same compute node, and its total data transfer dtv_W is denoted by (5).

$$dtv_W = \sum_{i=1}^{h} \sum_{j=1}^{h} dtv(w_i, w_j), i < j, \tag{5}$$

subjected to

$$dtv(w_i, w_j) = \begin{cases} 0 & \text{no data transmission between } w_i \text{ and } w_j, \\ E_r \cdot a & \text{otherwise,} \end{cases} \tag{6}$$

where $dtv(w_i, w_j)$ denotes the data transfer rate between w_i and w_j, and a denotes the number of task pairs communicated between w_i and w_j.

dtv_S denotes the total data transmission and will be calculated as follows.

$$dtv_S = dtv_P + dtv_W, \tag{7}$$

If we can minimize the amount of data transferred between compute nodes and between processes, we can largely reduce the communication overhead of the system and reduce the system latency.

$$Min(S_{cd}) \Leftrightarrow Min(dtv_S), \tag{8}$$

where S_{cd} represents the system communication delay.

Therefore, we can reduce dtv_P by assigning tasks to as few compute nodes as possible. In the selection of compute nodes, compute nodes are selected in descending order for task assignment based on the amount of available CPU resources in the compute nodes. This approach allows a single compute node to accommodate more tasks, and in addition to further reducing the amount of data transfer between compute

nodes, it also reduces system energy consumption by shutting down and hibernating idle compute nodes.

4 Ts-Stream Overview

Ts-Stream is a task scheduling strategy based on topology awareness. It is used to reduce the latency and energy consumption of distributed stream computing systems. This section focuses on a detailed discussion of Ts-Stream, including its system architecture, graph Laplacian-based graph partitioning approach, and task assignment strategy.

4.1 System Architecture

The Ts-Stream system adds a monitoring module, a database module, and a Ts-Stream scheduling generation module to the Storm system architecture, as shown in Fig. 3. The monitoring module regularly collects information from the system, such as the data transfer rate between executors, the load of executors and the load of worker nodes, and then stores them in the database. The Ts-Stream module reads this information from the database, first divides the different sub-topological graphs through the graph partitioning algorithm, and then generates a new schedule and delivers it to the Nimbus master node. To deploy our proposed task scheduling strategy, IScheduler will be implemented as an interface in Storm's custom scheduler.

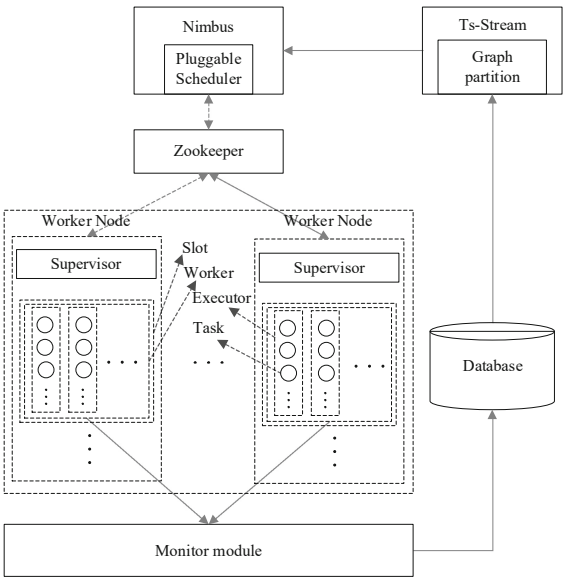

Fig. 3. Ts-Stream topology

In a distributed stream computing environment, the data stream rate and the load of the computing node are constantly changing. Moreover, due to the existence of a fault-tolerant mechanism, tasks may be restarted on other computing nodes. The existence of these problems will affect the performance of the system, so dynamic adaptive scheduling is essential. Ts-Stream will periodically check the operating status of the system. When the system is in a high-latency state for a long time or the remaining capacity of the computing node exceeds the threshold, it will generate a new schedule based on the topology information at this time.

4.2 Graph Division

Graph partitioning is used to divide the graph into two or k subgraphs to minimize the weights of the edges connecting different subgraphs and maximize the weights of the edges within the same subgraph. However, minimizing the value of cut edges can lead to some bad partitions, such as the partitioning of the topological graph into 1 vertex and $n - 1$ vertices in Fig. 4(a). In order to achieve a good partitioning as in Fig. 4(b), we can adjust the objective function, while making the sum of the edge weights in each part of the divided subgraphs as large as possible, the sum of the edge weights among the subgraphs is as small as possible. Using the Laplacian matrix, such segmentation results can be obtained simply and effectively.

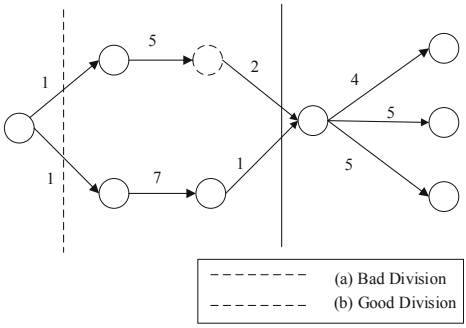

Fig. 4. Two different graph division results

In the graph partitioning phase, a graph partitioning method is described in Algorithm 1.

Algorithm 1: Graph partitioning algorithm based on graph Laplacian.

1. **Input**: $E = \{e_{1,2}, e_{1,3}, \ldots, e_{n-i,n}\}, i \in \{1, 2, \ldots, n\}$, k .
2. **Output**: k subgraphs after division, weights of cut edges.
3. **for** M **do**
4. $m_{ij} = e_{ij}$
5. **end for**
6. **for** D **do**
7. **if** $i = j$ **do**
8. $d_{ij} = \sum_{j=0}^{n-1} m_{ij}$
9. **else**
10. $d_{ij} = 0$
11. **end if**
12. **end for**
13. Calculate the eigenvalues and eigenvectors of L , Λ and Z_{nk}
14. $Z_{nk} = [y_1, y_2, \ldots, y_n]' \leftarrow Z_{nk} = [z_1, z_2, \ldots, z_k]$
15. **for** y_i **do**
16. Perform k-means clustering on y_i'
17. $class(y_i) \leftarrow class(y_i')$
18. **end for**
19. **return** k subgraphs and cut edge value.

The input of the algorithm includes the edge weight set $E = \{e_{1,2}, e_{1,3}, \ldots, e_{n-i,n}\}$, $i \in \{1, 2, \ldots, n\}$ of DAG and the number of subgraphs k to be divided. The output is the divided k subgraphs and the cut edge value. Steps 2 to 14 are to generate the adjacency matrix M and the degree matrix D according to the weights of the directed edges of the DAG. According to the result, the Laplace matrix L is solved, and the first k minimum eigenvalues $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ and the corresponding eigenvector $Z_{nk} = [z_1, z_2, \ldots, z_k]$ of L are solved. According to the Rayleigh-Ritz theory [23], k-means clustering is performed on the row vectors of the matrix composed of eigenvectors, and the results are mapped to the original graph to complete the division of DAG.

With this graph division method, adjacent tasks with high transmission rates can be allocated into the same node, and the amount of data transmission between computing nodes and between processes can be reduced, thereby realizing low-latency processing of the system.

4.3 Algorithm Description of Ts-Stream

In the task assignment phase, we propose a topology-aware task allocation strategy described in Algorithm 2.

Algorithm 2: Topology-aware task scheduling strategy.

Input: $_r$ and C_{cn}
Output: data stream task scheduling based on topology awareness.
$V = \{v_1, v_2, \ldots, v_n$
$E = \{e_{1,2}, e_{1,3}, \ldots, e_{n-i,n}$ $, i \in 1, 2, \ldots, n$
$G = \{G_1, G_2, \ldots, G_k \leftarrow G = V, E)$
Arrange C_{cn} in descending order.
If DAG $G = null \,||\, C_n = null$ **then**
    return null;
end if
while DAG $G! = null$ **do**
        Create a collection G_{ub};
        for $C_{cn_{\max}}$ **do**
            Add the two subgraphs connected by the cut edge with the largest current weight to G_{ub};
            while $C_{G_{sub}}$ $C_{cn_{\max}}$ **do**
                Add to G the subgraph with the largest weight of the edge connected to G_{ub};
            end while
            $C_{cn_{\max}} \leftarrow C_{G_{ub}}$;
        end for
        $G = null$;
end while
return topology-aware task scheduling

First, the data transfer rate between tasks is collected by the monitoring module to construct a DAG. Although there can be multiple tasks on an executor, in our experiments, there is only one task on an executor by default. Then submit the DAG and the parameter k of the number of subgraphs that need to be divided to the graph partitioning module. Where k by (9).

$$k = \frac{n}{w_e}, \tag{9}$$

where w_e takes the value of the number of executors in the worker.

The graph segmentation module divides the DAG into k parts and arranges all the cut edges in descending order according to the cut edge weights. The working nodes are also arranged in the order of capacity from largest to smallest. First, the two subgraphs connected by the edge with the largest weight are selected and put into the first working node, and then the subgraphs connected by the edge with the largest weight of its adjacent edges are also assigned to this working node until this working node reaches the set threshold C_a. The value of C_a by (10).

$$C_a = C_{cn} \cdot \alpha, \tag{10}$$

where C_{cn} is the available computational resources of the node and alpha is factor for resource utilization (set to 0.7 by default).

Then the subgraphs are assigned to other compute nodes in turn until all subgraphs are assigned. This allows the allocation to be done using the least number of compute nodes, and the rest of the nodes are dormant or shut down, which has resulted in energy saving. This also reduces the cross-node communication and reduces the system latency.

5 Performance Evaluation

In this section, we evaluate our Ts-Stream system. We first discuss the experimental environment and parameter settings, and then analyze the performance results.

5.1 Experimental Environment and Parameter Setup

Our proposed Ts-Stream system is developed based on Storm 2.1.0 and installed on CentOS 6.8. This cluster has 9 nodes, where 1 node is designed as a Nimbus node, 2 nodes are designed as Zookeeper nodes and the remaining 6 nodes are designed as Supervisor nodes. The Nimbus node uses DELL's R410, equipped with 12 Intel(R) Xeon(R) CPU X5650 @ 2.67 GHz 6-core processors and 12 GB of memory. Zookeeper nodes and Supervisor nodes are virtual machines, equipped with 2 Intel(R) Xeon(R) CPU X5650 @ 2.67 GHz 2-core processors and 4 GB of RAM. Each machine uses Storm 2.1.0 as the base system and is coordinated by Zookeeper 3.4.14. The software configuration of the Ts-Stream platform is shown in Table 2.

Table 2. Software configuration of the Ts-Stream

Software	Version
OS	CentOS 6.8 64bit
Storm	apache-storm-2.1.0
JDK	jdk1.8 64bit
Zookeeper	zookeeper-3.4.14
Python	python 2.7.2
Maven	Maven 3.6.2
MySQL	MySQL-5.1.73

We evaluate the system latency and system throughput by running WordCount and Top-N task topology. The task topology of WordCount and Top-N is shown in Fig. 5.

5.2 Performance Results

The experimental setting contains two evaluation parameters: system latency and system throughput.

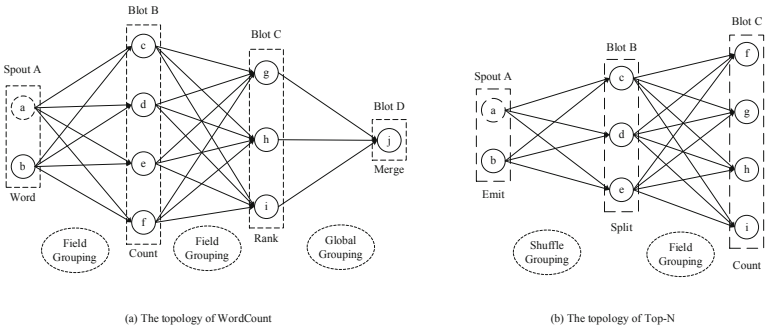

Fig. 5. The topology of WordCount and Top-N

(1) System latency

An important feature of stream computing systems is that they can process data in real time, so system latency is an important evaluation criterion. System latency is considered acceptable to users if it can be kept at the millisecond level. The lower the system latency, the better the real-time performance of the stream computing system. In Storm platform, the system latency can be obtained through Storm UI. We compare Ts-Stream with Storm's default scheduler. The processing latency metric is measured periodically over a period of 600 s.

In the WordCount experiment, the processing latency of the system fluctuated over time, but Ts-Stream's processing latency was lower than the default task policy of the Storm platform. As shown in Fig. 6, the processing latency of the TS-Stream policy and the default Storm policy are 1.91 ms and 2.72 ms, respectively. It is clear that the average system latency of Ts-Stream is smaller than the default Storm policy when the system is stable.

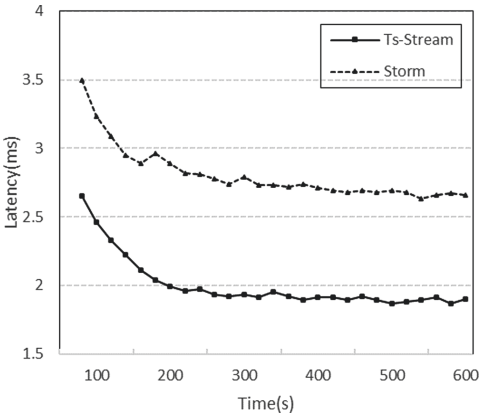

Fig. 6. System latency for running WordCount

When the system latency stabilizes over time, Ts-Stream has better system processing latency compared to the Storm platform. As shown in Fig. 7, the average system latency is 2.31 ms and 2.98 ms for Ts-Stream and Storm default task assignment policy, respectively, within [150, 600] seconds.

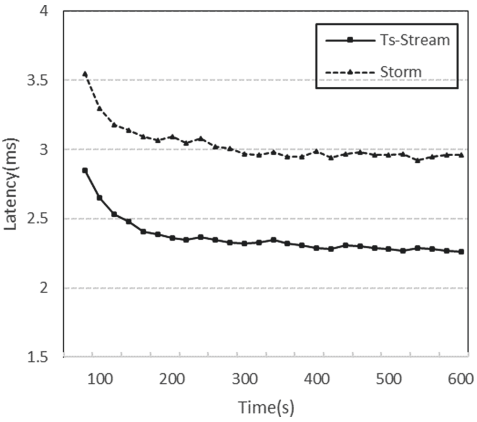

Fig. 7. System latency for running Top-N

(2) System throughput

System throughput reflects the system's ability to process data, which is estimated in terms of the number of tuples output by the DAG per second. The higher the system throughput, the better the stream computing system is able to process the data. In this set of experiments, we set the input rate of the data stream to 1000 tuples/s. The processing latency metric is measured periodically over 600 s.

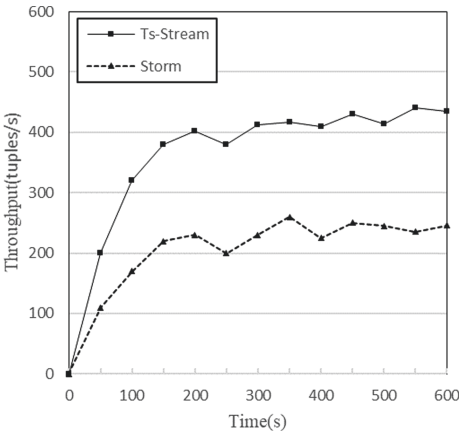

Fig. 8. System throughput for running WordCount

When the data transfer rate is kept stable, Ts-Stream has higher system throughput compared to the default Storm policy. As shown in Fig. 8, when the rate is set to 1000 tuples/s, the average system throughput of Ts-Stream and the default Storm policy during the stabilization phase is distributed as 412 tuples/s and 234 tuples/s. The average throughput of Ts-Stream proves to be higher than that of the default Strom policy.

As shown in Fig. 9, Ts-Stream has higher system throughput compared to the default policy of Storm when running the task topology of Top-N. During the stabilization phase of both policies, the average throughput of Ts-Stream and Storm's default policy are 405 tuples/s and 222 tuples/s, respectively.

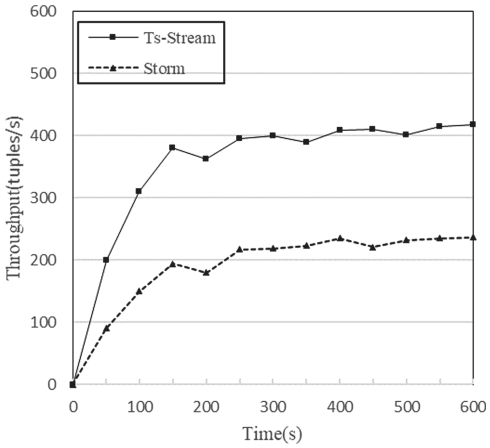

Fig. 9. System throughput for running Top-N

6 Conclusions and Future Work

We proposed a topology-aware task scheduling policy Ts-Stream. It uses a graph partitioning algorithm based on graph Laplacian to partition the task topology graph into *k* subgraphs with higher internal task communication. In the task scheduling phase, it minimizes the number of turned-on nodes by assigning tasks to the compute nodes with the highest capacity. Experimental results show that Ts-Stream performs significantly better than Storm's default scheduling, reducing system latency and improving throughput.

As part of future work, we will focus on the following areas:

(1) Consider other system resource constraints combined with the state of DAG vertices to further reduce the delay of the distributed stream computing system.

(2) Deploy Ts-Stream in the actual big data stream computing environment, such as intelligent recommendation system, real-time stock market analysis, embedded advertising and other scenarios.

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

References

1. Chintapalli, S., Dagit, D., et al.: Benchmarking streaming computation engines: storm, flink and spark streaming. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW, Chicago, IL, USA, pp. 1789–1792. IEEE (2016)
2. Shih, D., Hsu, H., Shih, P.: A study of early warning system in volume burst risk assessment of stock with big data platform. In: 2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis, ICCCBDA, Chengdu, China, pp. 244–248. IEEE (2019)
3. Kridel, D., Dolk, D., Castillo, D.: Adaptive modeling for real time analytics: the case of "Big Data" in mobile advertising. In: 2015 48th Hawaii International Conference on System Sciences, Kauai, HI, USA, pp. 887–896 (2015)
4. Sharif, A., Li, J., Khalil, M., Kumar, R., Sharif, M.I., Sharif, A.: Internet of things — smart traffic management system for smart cities using big data analytics. In: 2017 14th International Computer Conference on Wavelet Active Media Technology and Information Processing, ICCWAMTIP, Chengdu, China, pp. 281–284 (2017)
5. Storm Homepage. http://storm.apache.org/. Accessed 25 Apr 2021
6. Hadoop Homepage. http://hadoop.apache.org/. Accessed 25 Apr 2021
7. Farahabady, M.R.H., Samani, H.R.D., Wang, Y., et al.: A QoS-aware controller for apache storm. In: 2016 IEEE 15th International Symposium on Network Computing and Applications, NCA, pp. 334–342 (2016)
8. Liu, Y., Shi, X., Jin, H.: Runtime-aware adaptive scheduling in stream processing. Concurrency Comput. Pract. Experience **28**(14), 3830–3843 (2016)
9. Dongen, G., Poel, D.: Evaluation of stream processing frameworks. IEEE Trans. Parallel Distrib. Syst. **31**(8), 1845–1858 (2020)
10. Benjelloun, S., et al.: Big data processing: batch-based processing and stream-based processing. In: 2020 Fourth International Conference on Intelligent Computing in Data Sciences, ICDS, Fez, Morocco, pp. 1–6 (2020)
11. Aniello, L., Baldoni, R., Querzoni, L.: Adaptive online scheduling in storm. In Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems, pp. 207–218. ACM (2013)
12. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
13. Mehmood, E., Anees, T.: Challenges and solutions for processing real-time big data stream: a systematic literature review. IEEE Access **8**, 119123–119143 (2020)
14. Xhafa, F., Naranjo, V., Caballé, S.: Processing and analytics of big data streams with Yahoo! S4. In: 2015 IEEE 29th International Conference on Advanced Information Networking and Applications, Gwangju, Korea (South), pp. 263–270. IEEE (2015)

15. Liu, Y., Buyya, R.: Resource management and scheduling in distributed stream processing systems: a taxonomy, review, and future directions. ACM Comput. Surv. **53**(3), 1–41. Article No. 50. ISSN 0360-0300 (2020)
16. Govindarajan, K., Kamburugamuve, S., Wickramasinghe, P., Abeykoon, V., Fox, G.: Task scheduling in big data - review, research challenges, and prospects. In: 2017 Ninth International Conference on Advanced Computing, ICoAC, Chennai, India, pp. 165–173 (2017)
17. Peng, Y., Hosseini, M., Hong, H., Farivar, R., Campbell, R.: R-Storm: resource-aware scheduling in storm. In: Proceedings of the 16th Annual Middleware Conference, pp. 149–161. Association for Computing Machinery, New York, NY, USA (2015)
18. Fu, T., Ding, J., Ma, R., Winslett, M., Yang, Y., Zhang, Z.: DRS: dynamic resource scheduling for real-time analytics over fast streams. In: Proceedings 2015 IEEE 35th International Conference on Distributed Computing Systems, ICDCS, pp. 411–420. IEEE (2015)
19. Xu, J., Chen, Z., Tang, J., Su, S.: T-Storm: traffic-aware online scheduling in storm. In: 2014 IEEE 34th International Conference on Distributed Computing Systems, Madrid, Spain, pp. 535–544. IEEE (2014)
20. Zhang, Z., Jin, P., Wang, X., Liu, R., Wan, S.: N-Storm: efficient thread-level task migration in apache storm. In: 2019 IEEE 21st International Conference on High Performance Computing and Communications, pp. 1595–1602. IEEE (2019)
21. Eskandari, L., Huang, Z., Eyers, D.: P-Scheduler: adaptive hierarchical scheduling in apache storm. In: Proceedings of the Australasian Computer Science Week Multiconference, p. 26. ACM (2016)
22. Wei, H., Wei, X., Li, L.: Topology-aware task allocation for online distributed stream processing applications with latency constraints. Phys. A Stat. Mech. Appl. **534**, 122024 (2019)
23. Luxburg, U.: A tutorial on spectral clustering. Stat. Comput. **17**, 395–416 (2007)