Balanced Schedule on Storm for Performance Enhancement

Arwa Z. Selim[1], I. M. Hanafy[3]
Department of Mathematics and Computer Science
Faculty of Science, Port Said University
Port Said, Egypt

Noha E. El-Attar[2], Wael A. Awad[4]
Faculty of Computers and Artificial Intelligence[2, 4]
Benha University, Benha, Egypt[2]
Damietta University, Damietta, Egypt[4]

***Abstract*—In recent years, real-time and big data aroused and received a lot of attention due to the spread of embedded systems in almost everything in life. This has led to many challenges that need to be solved to enhance and improve systems that work on big real-time data. Apache Storm is a system used for computing and analyzing big real-time data of distributed systems. This paper aims to develop a scheduler to improve the scheduling of the applications represented by topologies on the Storm cluster. The proposed scheduler is hybridization between the scheduling algorithms of A3 Storm and the Workload scheduler. Its objective is to minimize the communication between tasks while balancing the workload on all cluster machines. The proposed scheduler is compared with the A3 Storm and Fischer and Bernstein's scheduling algorithm. The comparison has been made using four different topologies. The experimental results show that our proposed scheduler outperforms the two other schedulers in throughput and complete latency.**

Keywords—Real-time; big data; apache storm; scheduling

I. Introduction

Real-time applications such as IoT sensors, climate, and healthcare produce a large amount of continuous real-time data. The nature of this type of data is overgrowing where it can reach quintillions of bytes every day. This extreme and rapid growth of data leads to the term "big data" [1]. 5Vs features usually characterize big data; volume, variety, velocity, veracity, and value. Volume refers to the massive amount of data [2]. Variety means that there are different types of data that cause complexity. The rate at which the data is produced and transferred is the velocity, and it must be analyzed in real-time. Veracity is the precision level of data. Finally, the value is the valuable information derived from the data [1] [3]. Generally, the "big data" processing can be done through two processing techniques; batch and stream processing on high-performance computing resources [4]. Batch processing works on data that is previously stored. At the same time, stream processing refers to processing a large amount of data in real time. Big Data needs specified applications for processing the data, such as Hadoop for batch processing and Apache Storm, S4, Spark, and Flink for real-time streaming applications [4].

Real-time refers to the concept of time quantity, which implies the necessity for a real-time clock to measure it [5]. The real-time tasks are classified into three different cases: hard real-time, firm real-time, and soft real-time. The constraint in hard real-time tasks is to create results within specific time constraints or cause disastrous results. Firm real-time tasks also have to create the results before the specified time constraints, or the results will be invaluable. Soft real-time tasks have no time limitation, the results could be generated at any time, and it will be beneficial and acceptable [5] [6].

Now-a-days, the processing of streaming data is gaining more attention due to its sensitive cases. Thus, many researchers have tried to enhance the scheduling techniques to handle this huge amount of data and increase performance of processing (i.e., decreases the latency, increase the throughput, balance the network load, etc.). Most of the researchers' algorithm achieved those performance objectives. The relevant algorithms some of them achieved increment in the throughput, some achieved network load enhancement, others achieved decrement in latency, etc. Real-time scheduling for streaming data needs continuous improvements to get better results with better performance. So the issue here is to propose an algorithm that increases the performance of the processing of streaming data in using Apache Storm.

The main idea of the proposed algorithm is to reduce the communication between executors. The scheduler collects information during runtime then it creates a schedule using graph partitioning technique that partition the communication graph [7] [8]. At the end, the collected communication between executors during runtime is used and the pairs of most communicating executors are assigned to the same slot. This will achieve workload balance; improve resource utilization, high throughput with more reduced load on network.

The paper contribution is as follows:

1) We proposed a hybrid between two algorithms, the Workload scheduling algorithm and the A3 Storm algorithm, which improves the performance of the Apache Storm. This scheduler maximizes the throughput and minimizes the latency as the communication network load is reduced.

2) The scheduler is based on graph partitioning; its objective is workload balance and inter-executor communication.

3) Four topologies evaluate the scheduler, and the metrics of throughput and latency are collected as results.

4) A comparison is made against two alternative schedulers to find which has better results when running the topologies on them.

This paper is organized as follows. In Section 2, we introduce an overview of the Storm and its scheduling. The related work is shown in Section 3. Section 4 discusses the state-of-the-art scheduling algorithms. The proposed algorithm is discussed in Section 5. Section 6 discusses the results with comparison to the state-of-the-art algorithms. Section 7 is the conclusion.

II. Overview on Apache Storm

Apache Storm is a widely used real-time processing framework due to its capabilities of carrying out analytics on data streams with high throughput. Storm is an open-source real-time processing framework that contains several components: Topology, Nimbus, Slave, and zookeeper. [9]. Topology is considered the main component in Storm; it is a directed acyclic graph (DAG) that consists of spout and bolts [9] [10]. The spout is the primary source of a stream, and the bolt is the processing unit of the topology which handles the stream of data. Storm is deployed on a cluster that follows a master-slave model. The master is called the Nimbus node, which is responsible for organizing the topology and analyzing it [11][10]. It also distributes the tasks among the supervisor nodes and monitors any failure occurrence on them (i.e., if one of the supervisors fails, it redistributes the work among the remaining supervisors). The slave is any supervisor node. Storm can have one or more supervisors, and each supervisor can have one or more workers, which helps execute the tasks assigned by the supervisor. The worker also can have one or more executors which are responsible for running and executing the tasks. In the end, the task carries out the actual processing of the data (i.e., the tasks can be spouts or bolts). Also, Storm contains the zookeeper, which coordinates the work between the Nimbus and supervisors and saves their state (i.e., in case of Nimbus failure, it will restart from its last state as it is saved in zookeeper). Fig. 1 depicts the components of the Storm cluster.

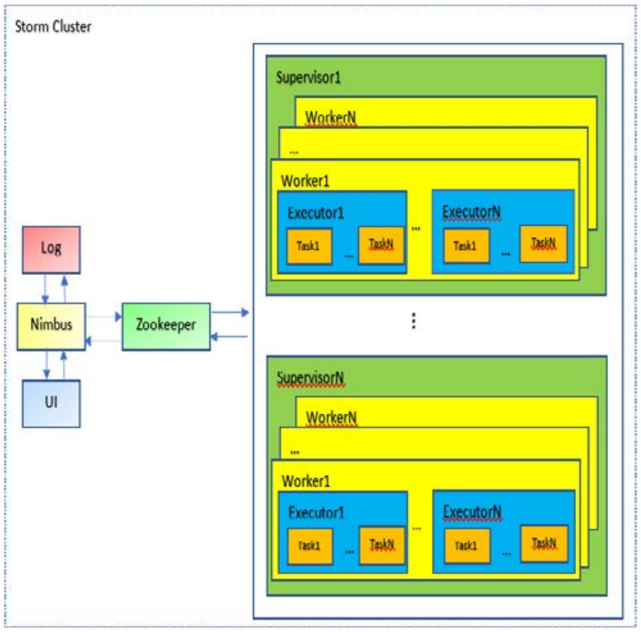

Fig. 1. Storm Cluster.

A default scheduler performs the standard scheduling process in Storm existed in Nimbus called EvenScheduler. This scheduler goes through two main phases; the first is assigning the executors to the workers, and the second is allocating the workers to the slots. EvenScheduler works based on round-robin strategy as follows [3]:

It iterates through the executors of the topology and allocates every executor evenly to the workers based on a round-robin algorithm.

The workers are allocated and assigned evenly to the supervisors, considering the available slots in each supervisor.

III. Related Work

One of the main benefits of Storm is that it is an open-source framework that allows creating custom schedulers that can meet the needs of the users and data. The default scheduler in Storm has some drawbacks which need to be improved. For instance, it evenly assigns the tasks to the cluster slots, but it does not consider the inter-node and inter-slot communication. Thus, the traffic may negatively influence the throughput and performance of the processing. Recently, many researchers have proposed enhanced scheduling algorithms that can improve the performance of Storm.

Aniello L. et al. [3] have proposed two scheduling algorithms, offline and online. The offline scheduling algorithm is based on the topology structure and how its components are interconnected with each other. It used the round-robin algorithm to assign slots to nodes. The online scheduling algorithm was based on monitoring the communication and the performance of the system at run time. It monitored the traffic of exchanged tuples between executors, sorted the executors in descending order according to the communication patterns, and assigned the most communicating executors in the same slot. Then the communication pairs of workers are iterated in descending order and assigned to the nodes. This algorithm reduced the inter-node traffic and communication, which affected the network load and the throughput.

Xu J. et al. [12] have developed a traffic-aware online scheduling algorithm that can reduce the inter-node and inter-process traffic by monitoring the traffic and workload information during runtime. It expedites the data processing by using the traffic-aware online algorithm for assigning executors. It also enables fine-grained control over worker node consolidation to obtain better performance while using fewer worker nodes.

In the same context, Peng B. et al. [11] have presented a resource-aware scheduling algorithm to improve resource utilization and reduce network latency. This algorithm is based on assigning the task according to an improved breadth-first traversal algorithm. It allocates the node ports that conform to the resource constraints and the network distance requirement.

Fischer L. and Bernstein A. [8] have proposed a workload scheduling based on the graph partitioning technique. It works during runtime and collects the behavior of communication of the topologies that are running. Then it partitions the communication graph to produce the schedule using software called METIS graph partitioning. METIS uses a multilevel graph bisection. It makes a miniature version of the graph by coarsening it and collapsing the nodes and edges. Then it partitions the resulting small graph before un-coarsening it to its first form. It adapts the partition at each step of the un-coarsening to consider the newly un-collapsed edges and vertices. This scheduler improves resource utilization, reduces the network load, and increases the throughput.

Another direction is presented by Li C. et al. [13]. They have developed a Storm topology dynamic optimization strategy (STDO-TOC) as a real-time scheduling algorithm. The STDO-TOC uses bolts capacity and analyzes the message queue congestion degree to alter the performance parameters during runtime. If the topology bottlenecks are found, they are automatically removed. This leads to optimizing the topology dynamically.

Another resource-efficient algorithm for streaming application scheduling D-Storm has been presented in Liu X. and Buyya R. [14]. D-Storm tracks the streaming tasks during runtime to collect resources and communications and use them in the scheduling process to pack the communicating tasks compactly. This strategy of tight scheduling reduces resource utilization and inter-node communication.

Muhammad A. et al. [15] have developed a topology-based resource-aware scheduling algorithm called Top Storm. It is based on finding the most communicating executors and putting them closer to each other to reduce the number of nodes used to execute topology. Top-Storm is considered a topology-based as it looks at the DAG of the topology to find the connections between the executors. Also, it is resource-aware as the assigning of executors is made based on the computation power of nodes.

An enhanced version of Top Storm called A3 Storm has been developed by Muhammad A. and Aleem M. [4]. It works offline by finding the most communicating executors from the DAG of the topology and putting them closer to each other, and assigning them to the nodes according to the most powerful one. At the same time, it can work online by using traffic beside the topology structure. It reads the inter executor traffic, sorts it into descending order, and then assigns it to the most influential nodes. This scheduler improves resource utilization and increases the throughput of the topology. Finally, Table I displays a brief review of the above-mentioned scheduling algorithms.

TABLE I. A Brief Review of Real-Time Scheduling Algorithms

Scheduling Aspects	**Scheduling Algorithms Characteristics**						
	Resource Aware	***Traffic-Aware***	***Dynamic***	***Heterogenous***	***Self-Adaptive***	***Topology Aware***	***Network-Aware***
Aniello, et al., 2013) [3]	x	✓	✓	✓	✓	✓	x
Xu, et al., 2014 [12]	x	✓	✓	✓	✓	x	x
Peng, et al., 2015 [11]	✓	x	x	✓	x	x	✓
Fischer & Bernstein, 2015) [8]	✓	✓	✓	✓	✓	x	✓
Li, et al., 2017 [13]	✓	✓	✓	✓	✓	✓	x
Liu & Buyya, 2017 [14]	✓	✓	✓	✓	✓	x	x
Muhammad, et al., 2021 [15]	✓	x	x	✓	x	✓	x
Muhammad & Aleem, 2021) [4]	✓	✓	✓	✓	✓	✓	x

IV. Preliminaries

A. Workload Scheduler

The workload scheduler is the standard storm topology. It has two types of views, logical view, and physical view. The logical view, $T = (N, C)$, consists of N spouts and bolts connected with C number of connections. While the physical view of storm topology is represented by graph $G = (V, E)$, where V represents the vertices, and E represents the edges of the graph. A set of task instances represents the spout and bolts $v_i \in V$, and $|v_i| = d_i$ represents the degree of parallelism of each component, spout, or bolt. Every two sets of vertices V are connected by one edge E. Generally, the graph is weighted as follows [8]:

- The vertex weights are represented by the sum of the number of all released and received tuples.
- The edge weights are the number of messages released from any of the spout or bolt instances.

The main idea in graph partitioning is to partition the vertices into equal partitions to reduce the edges' number connecting the vertices of different partitions based on the k-way partitioning method. To clarify the k-way partitioning, if a given graph $G = (V, E)$, the vertices will be partitioned into M number of partitions P, where M is equal to the supervisor machines number in the cluster. Where $\bigcup_{m=1}^{M} P_m = V$ and $\bigcap_{m=1}^{M} P_m = \emptyset$. [8] [16].

The communication between partitions can be represented as a matrix. If there is a task τ_u and partition P_a. To check if the task τ_u is assigned to the partition P_a:

$$M_{\tau_u,P_a} = \begin{cases} 1, & if\ the\ task\ \tau_u\ is\ assigned\ to\ the\ partition\ P_a. \\ 0, & if\ the\ task\ \tau_u\ is\ not\ on\ the\ partition\ P_a. \end{cases} \quad (1)$$

Then the communication between the nodes in the communication graph can be represented as follows in (2) [8]:

$$C_{\tau_u,\tau_v} = \sum_{a=1}^{P} \sum_{b=1}^{P} M_{\tau_u,P_a} \times M_{\tau_v,P_b} \quad (2)$$

Where M is the total number of partitions.

According to (2), the formula of the node's communications can be summarized as follows:

$$C_{\tau_u,\tau_v} = \begin{cases} 1, & if\ task\ \tau_u\ and\ task\ \tau_v\ are\ on\ different\ partitions \\ 0, & if\ task\ \tau_u\ and\ task\ \tau_v\ are\ on\ the\ same\ partitions \end{cases} \quad (3)$$

Finally, the cost function of the partitioned graph G, which is partitioned into M partitions, can be defined as follows [8]:

$$cost(G, M) = \sum_{u=1}^{|V|} \sum_{v=1}^{|V|} C_{\tau_u,\tau_v} \times e_{u,v} \quad (4)$$

where |V| is the total number of vertices of the graph and $e_{u,v}$ represents the edge weights.

This partitioning algorithm aims to optimize the costs for the partitions. In this algorithm, there is another constraint: the partitions should be balanced for the workload. The load balance factor L_{im} is defined as below [8]:

$$L_{im}(G, M) = max({}^{P_i}/_{AP}) \quad (5)$$

where P_i is the sum of weights of all vertices in partition i, and AP is the average weight of partition over all partitions.

The workload scheduler uses METIS as software for graph partitioning, which is used for partitioning a graph into equal partitions [16]. The partitioning process includes three main phases: coarsening, partitioning, and un-coarsening.

Initially, the coarsening phase reduces the size of the graph by combining a set of vertices into one single vertex. The weight of this single vertex must be equal to the sum of weights of all vertices combined in it.

Then the partitioning phase is done on the coarsest graph to receive balanced partitions with respect to the workload.

Finally, the un-coarsening phase is used to return to the original graph and refine the resulting partitions.

To assign the partitions, their number must be equal to the number of machines in the cluster. Each partition will be assigned to one supervisor, and its tasks will be assigned to the slots. Finally, the executors in each partition are assigned to slots in a round-robin manner [7]. A flowchart of the workload scheduler is illustrated in Fig. 2.

B. A3 Storm Scheduler

A3 Storm scheduler is a topology, traffic, and resource-aware scheduler. It can find the connections between the executors by considering the DAG of topology, so it is considered a topology-aware scheduler. It also puts inter-executor communication into consideration, so it is a traffic-aware scheduler. Finally, it performs the physical mapping according to the computation power of nodes; thus, it also resources aware scheduler [4].

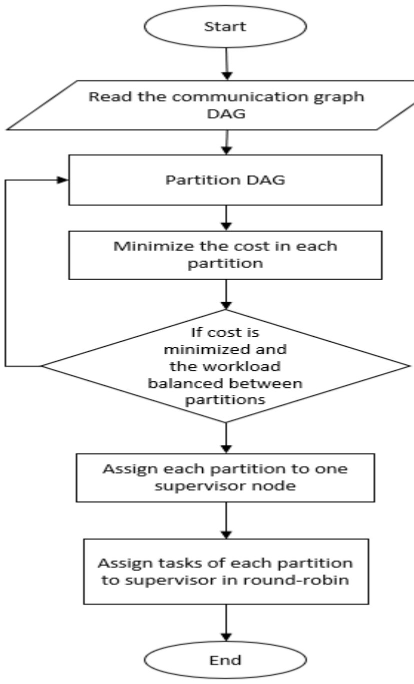
Start
Read the communication graph DAG
Partition DAG
Minimize the cost in each partition
If cost is minimized and the workload balanced between partitions
Assign each partition to one supervisor node
Assign tasks of each partition to supervisor in round-robin
End

Fig. 2. Workload Scheduler Flowchart.

In general, the A3 Storm scheduler follows two steps: (1) Assignment of Executors, either by using the DAG or the traffic of topology. In this step, the scheduler initially gets the number of worker processes and the inter-executor traffic for executors unassigned from the traffic log. Then the executor assignment process begins by sorting the executors in descending order according to the inter-executor traffic; after that placing the most communicating executors as close as possible to each other. (2) Assignment of Slot; in this step, the created groups of the highest communicating executors are assigned slots. Then these slots are assigned to nodes which are sorted in descending order according to the most computationally powerful node calculated by (6) and (7). [4] [15].

$$Computation_{Power} = \propto \times\, (Speed) + (1 - \propto) \times RAM \qquad (6)$$

$$Speed = no.of.Cores\ \times no.of.Sockets \times frequency \times no.of.Flops \qquad (7)$$

Where, $\propto$ is an adjustment factor equal 0.8.

V. Formulation of the Balance Workload Storm Scheduler Algorithm

The proposed algorithm hybridizes the Workload scheduler [8] and the A3 Storm scheduler [4]. As mentioned before, the Workload scheduler aims to reduce the network utilization by reducing the inter-node communication to increase the overall throughput and balance the workload among all available machines in the cluster. At the same time, the objective of the A3 Storm scheduler is to reduce the inter-node traffic, increase the throughput, and improve resource utilization.

The proposed algorithm of Balance Scheduling on Storm (BSS) is based on combining both the Workload scheduler and A3 Storm to enhance the performance of the Workload scheduler in increasing the data throughput and reducing the time latency. The main idea of the BSS is to apply the methodology of workload scheduler to monitor the metrics of Storm, collect the data of the communication graph during runtime, partition the graph, and balance the workload among the available machines in the cluster. Then, we have replaced the last step in the Workload scheduler (i.e., assigning each partition to one node and assigning tasks to slots in a round-robin strategy) by the A3 storm first phase (i.e., monitoring the communication between tasks). Therefore, after assigning the partitions to the nodes, the tasks will be assigned to the slots according to their inter-executor communication, which means that the most communicating executors will be assigned to the same slot, which will reduce the network communication between slots and in result this will reduce the latency and increase the throughput.

A. *The Proposed Algorithm of Balance Scheduling on Storm (BSS)*

The proposed algorithm of Balance Scheduling on Storm is based mainly on managing the execution of the topology. Its input and output are the unassigned topologies and the executor to node assignment, respectively. The phases of the proposed BSS algorithm can be concluded as follows:

1) The scheduler collects information about the topology, where it gets the number of worker processes (slots) required to execute the topology. Then it obtains the inter-executor connection of the unassigned executors and the number of the unassigned executors.

2) Calculating the maximum number of executors required per slot by (8):

$$e_{es} = \frac{total\_number\_of\_executors}{total\_number\_of\_slots\_required\_by\_topology} \qquad (8)$$

3) Finding the available nodes in the cluster.

4) Partitioning the vertices of the graph into equal partitions using the k-way partitioning method, where the number of partitions should be equal to the number of the topology workers.

5) Sorting the nodes where the nodes with no workers are put at the beginning, and the partially busy nodes are put at the end. In this stage, the scheduler has to ensure that the number of nodes is not smaller than the number of partitions.

6) Sorting the tasks in each pair in descending order according to their inter-executor traffic.

7) Assigning tasks to the slots, where the most communicating tasks are assigned to the same slot until it reaches the maximum number of tasks assigned to it. Then assigning the next task to the next slot and so on until all partition tasks are assigned to the slots of the node. The pseudo-code of the proposed algorithm of Balance Scheduling on Storm is shown in Algorithm 1.

Algorithm (1): Balance Scheduling on Storm

```
1.      function Schedule (U);
Input: Unassigned Topologies U
Output: Node-Executor assignment
2.      for each topology $u_i \in U$ do
3.        $n = u_i.numOfWorkers$;
// get the number of worker processes
4.        $e_{un} = u_i.UnassignedExecutors()$;
// get the list of Inter-Executor for unassigned executors
5.        $e_{total} = e_{un}.Count()$;
6.        $e_{es} = ceil(e_{total}/n)$


7.        Nodes = cluster.getAvailableNodes( );
8.        partition_file = graph.part($Nodes$);
// partition = number of workers
9.        Nodes.sort( );
// Nodes that have no worker are put at the beginning and
        partially busy nodes are put at the end
10.       $slots_{as}$ = 0;
11.       if Nodes.size( ) >= partition.size( ) do
12.         Foreach $n \in Nodes$ do
13.          Foreach $tasks \in partition.tasks$ do
14.            tasks.sort("Desc");
// task pairs are sorted in descending order according to the
        InterExecutor traffic
15.            While tasks != null
16.              For each task ∈ tasks
17.                If Count <= $e_{es}$
18.                  mapExecutorToSlot($slots_{as}$, task);
// Assign most communicating task pairs to the same slot;
19.                  Count ++;
```

```
20.                     End If
21.                     $slots_{as}$ ++;
22.                   End for
23.                 End While
24.              End for
25.            End for
26.          end If
27.        End for
28.        End
```

Also, all the utilized notations and functions in the algorithm are described in Table II.

TABLE II. LIST OF NOTATIONS

Symbol	Definition
U	Unassigned topologies
N	Total number of workers required for the execution of the topology
e_{un}	List of all unassigned executors of a topology
e_{total}	Total number of topology executors
e_{es}	The maximum number of executors per slot.
Graph.part(n)	Partition the topology into partitions equal to the number of workers of the topology
Cluster.getAvailableNodes()	Get the available nodes in the cluster
Nodes.size()	Total number of nodes in cluster
Partition.size()	Total number of partitions
Partition. Tasks	The list of tasks of each partition
MapExecutorToSlots	Assign tasks to slots

VI. EXPERIMENTS AND RESULTS

The experimental study is done on the Apache Storm cluster, which has a Nimbus node, Zookeeper node, and two supervisor nodes having three and four slots, respectively. The configuration of the first supervisor is as follows; it has Ubuntu 20.0.1 LTS 64-bit installed on it with GNOME version 3.36.3, with “10.6 GB” memory, Intel®CoreTM i7-4810MQ CPU @ “2.80GHz × 2” processor, and “536.9 GB” disk capacity. The second supervisor has Ubuntu 14.04 LTS with OS type 64-bit with “9.5 GB” memory, Intel®CoreTM i7-4810MQ CPU @ “2.80GHz” processor, and disk capacity “21.7 GB”. Both of the supervisors have 1000 Mb/s network connectivity speed. Each supervisor has Apache Storm 1.1.1, Apache Zookeeper 3.5.7, and Java Open JDK 13. The proposed algorithm of BSS is compared with the default Workload scheduler and the A3 Storm [4] on four benchmark topologies:

1) *SOL topology [17]:* It has a chain-like structure. It has a spout and a set of bolts. It loads its data directly from the data source. Random messages are used to create sentences with words’ lengths specified by the user. It consists of one spout and a user-defined number of bolts. This topology aims to trace the network's performance, so it is better to keep the computation as minimum as possible.

2) *Rolling count topology [17]*: It applies rolling counts of incoming terms. By the term rolling, it uses a sliding window to trace the statistics of a term until the current window compared to the one in previous. A rolling count tuple per term is emitted by reaching the end of each time window, and it consists of the term. The term’s rolling count is a metric that points at how this term is trending now and the actual duration of the sliding window. The term can be emitted from more than one node, so a bolt must join and rank the terms. So, it consists of a spout that directly loads the data from the data source, a bolt that splits the sentences, and a rolling count bolt that uses field grouping to count the terms and group them to emit the ranks of each term.

3) *Word count topology [18]:* The spout emits streams of sentences and sends them to a bolt that splits these sentences into words and emits them to another bolt that, using field grouping, can count how many times each word has occurred. Field grouping means that based on the value of the word, the same word must always go to the same instance so that it can be counted.

4) *Spike detection topology [18]:* The spout receives a stream of data from sensors and emits them to bolts to monitor the occurrences of values that have spikes. Spout emits this stream to a bolt named Moving Average, which gets the data grouped according to the IDs of the device. When a new stream of data is received, the bolt aggregates the new values of a device to the list of values of the same device and emits new events consisting of the device ID, its current value, and its values moving average. These emitted events go to another bolt named Spike Detection, as it detects if there is a spike in the current event or not.

B. *Results and Discussion*

To evaluate the performance of the proposed BSS algorithm, we consider two performance metrics:

1) *Throughput*: represents the number of tuples processed per unit time [18].

2) *Complete Latency*: is the average time a tuple takes to be entirely processed by the topology [18].

The experiment has been done by applying the proposed BSS, Workload scheduler, and A3 Storm scheduler, the four benchmark topologies mentioned above. Each algorithm has been run three times to get the average results for the three compared algorithms.

3) *SOL topology results*: SOL is generated with one spout and two bolts. The required number of workers is two, and the number of executors and tasks equals a value of nine. The results are depicted in Table III.

As shown in Table III, the Balance scheduling Storm algorithm achieved the highest value of throughput, which was “4059.67 tuples/second” at the second 240. In comparison, the most negligible value of throughput was “15.33 tuples/second” and achieved by the Workload scheduler after “60 seconds”. It is also obvious that the BSS had the best results for the throughput than the two other algorithms till “600 seconds”. At the “660 seconds” and “720 seconds” the A3 Storm showed better throughput results than the BSS and the Workload scheduler algorithms, then starting from the second 780, the

BSS returned to give high throughput than the two other schedulers. Regarding the latency, it is found that the best complete latency was "63.833 milliseconds" for the BSS the first "60 seconds", while the worst complete latency value was 949, which is recorded by the Workload scheduler after "120 seconds". The BSS gave the best latency results during the overall execution except at the second 180; the Workload scheduler gave the best latency. Finally, by looking at the average in Table III, it is obvious that the BSS gave better average results in terms of throughput and complete latency.

TABLE III. SOL TOPOLOGY EVALUATION RESULTS

Time (sec.)	**Throughput (tuples/sec.)**			**Complete Latency (Millisecond)**		
	Workload Scheduler	***A3 Storm***	***The Proposed BSS***	***Workload Scheduler***	***A3 Storm***	***The proposed BSS***
60	15.33	84	**183.67**	437.50	77.50	**63.833**
120	867	2045	**1647.67**	949	764.10	**159.33**
180	1364	3384.33	**3194**	**443.80**	771.80	859.47
240	1623	2888	**4059.67**	387.83	756	**383**
300	1136.67	2104.67	**3485**	401.23	747.67	**251.13**
360	538.67	1585.33	**2609.67**	442.50	794.30	**225.97**
420	453	1101	**3333.33**	515.13	797.30	**197.43**
480	212.67	987.33	**1955.33**	539.63	699.23	**192.63**
540	422.33	910	**1871.33**	549.93	604.87	**186.50**
600	637.33	1298.33	**1880.67**	544.73	489.63	**185.07**
660	987.67	**2072.67**	1865	520.07	391.30	**183.37**
720	1410	**1926**	1804.67	484.20	349.80	**184.33**
780	2086.33	2044.33	**2781.33**	445.93	315.27	**181.33**
840	1769	2419.33	**3689.33**	419.10	291.93	**168.97**
900	2123.67	2206.67	**2959**	382.53	276.97	**162.23**
Average	1043.11	1803.8	2487.98	497.54	534.2	238.973

According to the reported results, it is clear that the BSS outperforms both workload scheduler and A3 Storm in both throughput and complete latency according to the overall average values. In contrast, the Workload scheduler and A3 Storm are recorded the worst average on throughput and the complete latency, respectively. Fig. 3 and Fig. 4 depict the comparison between the three algorithms' throughput and complete-time latency results.

4) Rolling count topology results: Rolling Count topology is generated on one spout and two bolts with four workers and 26 executors and tasks. The experiments using the three compared schedulers are presented in Table IV and depicted in Fig. 5 and Fig. 6.

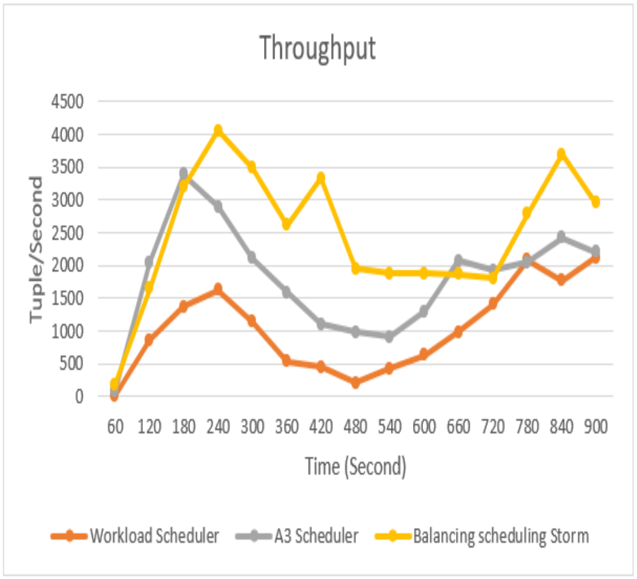

Fig. 3. Throughput Comparison Result between Three Schedulers for SOL.

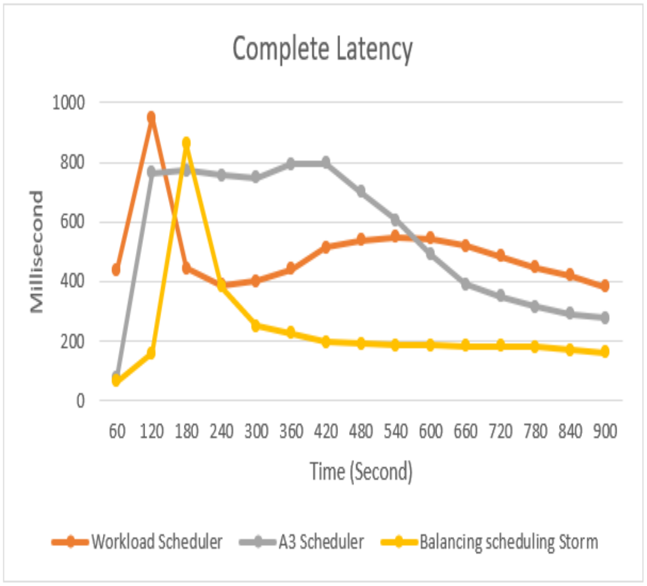

Fig. 4. Complete Latency Comparison Result between Three Schedulers for SOL.

TABLE IV. SOL TOPOLOGY EVALUATION RESULTS

	Throughput (tuples/sec.)			**Complete Latency (Millisecond)**		
Time (sec.)	***Workload Scheduler***	***A3 Storm***	***The proposed BSS***	***Workload scheduler***	***A3 Storm***	***The proposed BSS***
60	0	0	**43.67**	0	0	791.38
120	3757.33	2647	**7978.33**	828.46	**424.51**	500.12
180	7171.33	4652.33	**11547.33**	693.91	473.84	**406.36**
240	9029	5422.33	**13006**	650.52	976.38	**366.88**
300	**12457.67**	5427.67	11772	501.28	989.39	**374.19**
360	11976	4284	**12162**	452.36	1123.46	**365.63**
420	14029	3610.33	**14039.33**	453.09	1295.57	**360.11**
480	**15489.33**	2226.67	12862.33	471.16	1507.76	**348.85**
540	13130.33	2218	**17422.33**	502.71	1740.99	**337.25**
600	12256.67	3152	**17847.33**	562.91	1844.03	**330.63**
660	11614.67	5202	**15966.33**	610.09	1727.88	**331.03**
720	10104.67	7001.33	**14791**	663.46	1605.83	**332.59**
780	8298.67	7322.67	**17422.33**	708.77	1541.14	**328.19**
840	11112.67	8103.67	**18629**	725.97	1527.58	**322.37**
900	12185.33	7801.67	**24627.33**	712.18	1511.18	**310.48**
Average	10174.18	4604.78	**14007.78**	569.12	1219.3	**387.07**

Table IV shows that the BSS has the highest average of throughput, and the A3 Storm has the least average value of throughput. The best value reached of the throughput was "24627.33 tuples/second" at the second 900 by the BSS. During the overall execution the BSS showed better results for throughput. But only at "240 seconds" and "480 seconds", the BSS gave lower results and the Workload scheduler showed higher results than the BSS. For the complete latency values, the results showed that the BBS has a minor average of complete latency. In comparison, the Workload scheduler has the highest value of the average complete latency. The least value recorded was "310.48 milliseconds" for the BSS at the second 900, and the highest value was for the A3 Storm "1844.03 milliseconds" at the second 600. Furthermore, as displayed in Table IV, at the first "60 seconds," both the Workload scheduler and A3 Storm could not finish any data processing, and their recorded throughput was zero. While the BSS processed the data in this limited time and produced "43.67 tuples/second". At the end of the Table IV, it is shown that the BSS had the best average results for the throughput and complete latency.

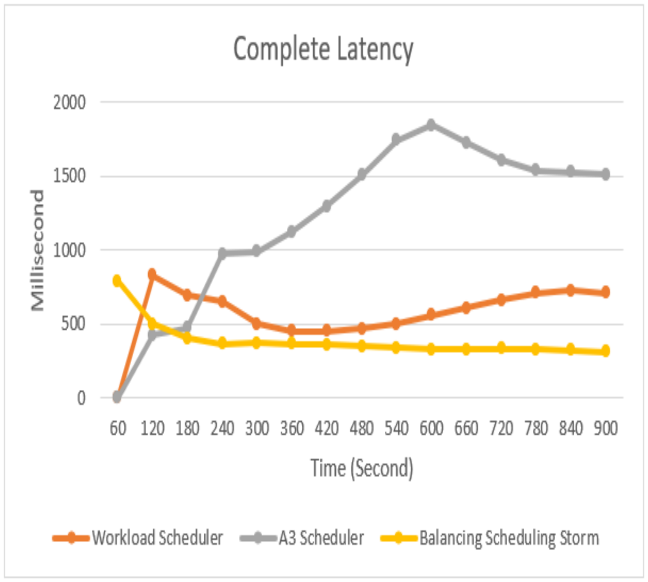

Fig. 5. Complete Latency Comparison Result between Three Schedulers for Rolling Count.

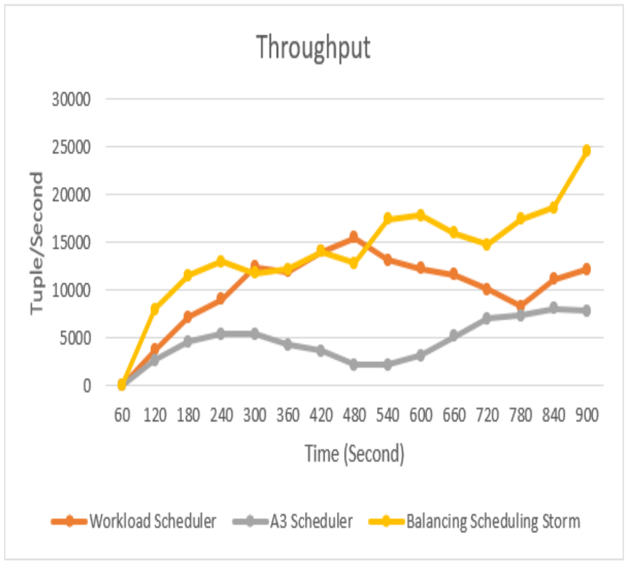

Fig. 6. Throughput Comparison Result between Three Schedulers for Rolling Count.

5) Word count topology results: This topology is executed using two workers with one spout and two bolts. It also contains 13 executors and tasks for the “3.8 GB” dataset size.

TABLE V. WORD COUNT TOPOLOGY EVALUATION RESULTS

	Throughput (tuples/sec.)			**Complete Latency (Millisecond)**		
Time (sec.)	***Workload scheduler***	***A3 Storm***	***The proposed BSS***	***Workload scheduler***	***A3 Storm***	***The proposed BSS***
60	417.33	344.67	**2990.33**	296.67	229.63	**146.87**
120	13543	11708.67	**22176**	96.2	233.87	**84.57**
180	27664.33	20740.33	**32432.67**	60.87	140.53	**52.7**
240	26123.33	22706	**29887.33**	50.97	104.63	**46.93**
300	24649	23835	**27501.67**	46.6	89.83	**44.17**
360	22638.33	19374.33	**32028**	**44.1**	86.03	44.73
420	25167	15821.33	**28415.33**	**42.43**	87.27	47.73
480	24097.33	14178.67	**30859.67**	**40.63**	91.33	48.6
540	22573	9098.33	**33317.67**	**39.3**	97.63	50
600	25178.67	8467.33	**29689.33**	**37.9**	104.77	49.93
660	24579	8629.33	**33404**	**36.77**	109.3	49.1
720	12448.33	13077.67	**33584.33**	**35.53**	110.77	48.43
780	19160	16323.67	**27318.33**	**35.57**	109.1	48.47
840	18233	16929.67	**29635.67**	**35.53**	105.03	49.4
900	18468	21994.67	**30709.67**	**35.4**	99.73	50.63
Average	*20329.31*	*14881.98*	***28263.33***	*62.3*	*119.96*	***57.48***

Table V shows that the BBS recorded the highest throughput, while the A3 Storm recorded Storm the least. The BSS has reached the maximum value of throughput, which is “33584.33 tuples/second” after “720 seconds,” and the A3 Storm has the least value of throughput, which is “344.67 tuples/second” after “60 seconds”. Regarding the complete latency, the experiments' results showed that the least complete latency value was “35.4 milliseconds” by the Workload scheduler after “900 seconds” of execution. But in contrast, the Workload scheduler recorded the highest value of complete latency, which was “296.67 milliseconds” after the first “60 seconds”. For the BSS, the reported average complete latency value was the least. From the comparison presented in Fig. 7 and Fig. 8, it is obvious that the Balancing Scheduling storm enhanced the system's performance.

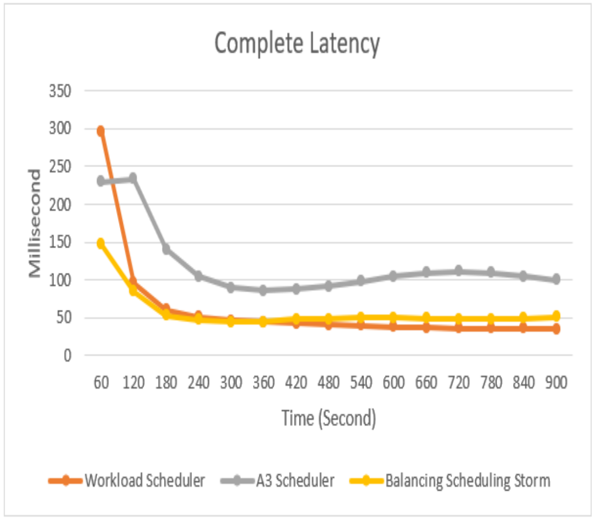

Fig. 7. Complete Latency Comparison Result between Three Schedulers for Word Count Topology.

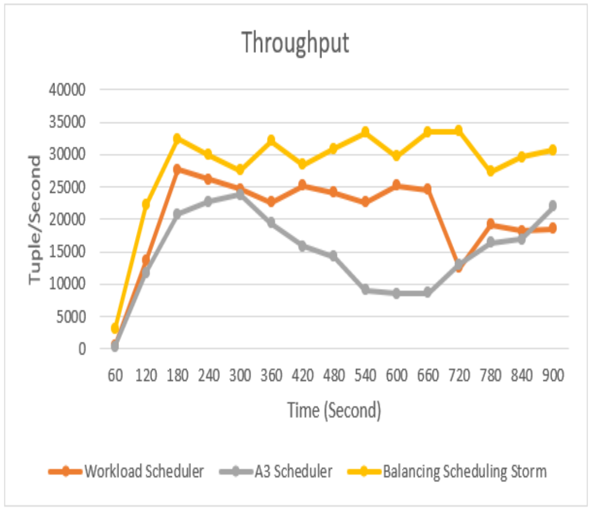

Fig. 8. Throughput Comparison Result between Three Schedulers for Word Count Topology.

6) Real-time spike detection topology evaluation: Other experiments are carried out based on the real-time spike detection topology described in the Intel Lab Data [19]. The data set presented here is collected from 54 sensors in the Intel Berkeley Research lab. The data set included about two million readings gathered from these sensors. The topology has been deployed with a dataset about "11 GB" size generated by the repetition of the data presented in Intel Lab Data [19]. The recorded results after running the three schedulers are recorded in Table VI, and the comparison between these results are depicted in Fig. 9 and Fig. 10 are gained.

TABLE VI. SPIKE DETECTION TOPOLOGY EVALUATION RESULTS

	Throughput (tuples/sec.)			**Complete Latency (Millisecond)**		
Time (sec.)	***Workload scheduler***	***A3 Storm***	***The proposed BSS***	***Workload scheduler***	***A3 Storm***	***The proposed BSS***
60	5.33	0	**55.67**	1866.24	**0**	730.40
120	**998.67**	728.33	981.33	**797.75**	1405.14	936.81
180	1383.33	**1554.33**	1303.33	710.82	**607.35**	697.87
240	1486	**2358.67**	1955	714.35	**484.22**	513.49
300	1269	1462.67	**2036.67**	753.16	465.48	**452.48**
360	1067	1842.33	**1991.33**	821.36	476.09	**437.50**
420	954	1284.67	**2311.33**	824.58	475.98	**448.23**
480	902.67	903.33	**1740.33**	813.78	531.24	**481.70**
540	671.33	1143.67	**1270.67**	789.39	524.58	**515.08**
600	1006	999	**1578**	786.27	535.59	**488.90**
660	745.67	1357.67	**1746.67**	778.69	525.92	**473.79**
720	920	1397.33	**2205.67**	776.01	515.26	**462.69**
780	1141.67	1643.33	**1747.33**	784.31	504.56	**461.25**
840	1486.67	1553.33	**1889.33**	786.86	503.45	**444.81**
900	2056.67	1630	**2269.33**	709.07	501.22	**427.87**
Average	1072.93	1323.91	**1672.13**	847.51	537.07	**531.53**

Table VI shows the comparison results between the three algorithms in terms of throughput and complete latency. The Balancing Scheduling Storm has the maximum value of the average throughput. In contrast, the Workload scheduler has the minimum value. The best result of the throughput is "2311.33 tuples/second", which was reached by the Balancing scheduling storm after "420 seconds". The smallest value was "0 tuples/second," which was reached by the A3 Storm after "60 seconds". The next worst value of throughput was "5.33 tuples/second" at the first "60 seconds" and was reached by the Workload scheduler.

Also, the latency results are described in Table VI. The worst latency value was for the Workload scheduler at the first minute; it was "1866.24 millisecond". The best value was for the Balancing Scheduling storm after "15 minutes" it reached the value of "427.87 milliseconds". And by looking at the end of the Table VI, it is obvious that the BSS had the best average results in both terms, the throughput and the complete latency.

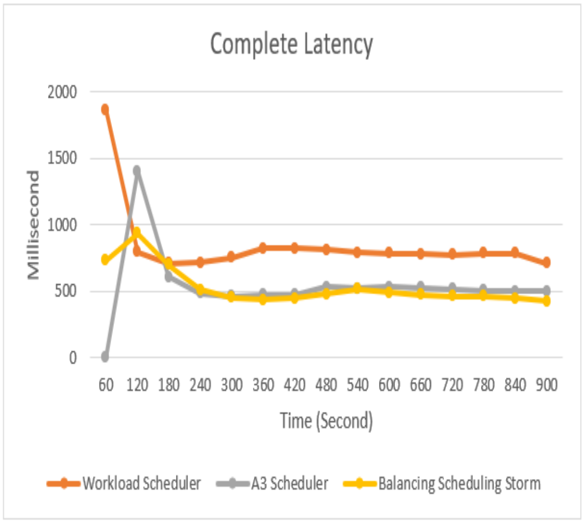

Fig. 9. Complete Latency Comparison Result between Three Schedulers for Spike Detection Topology.

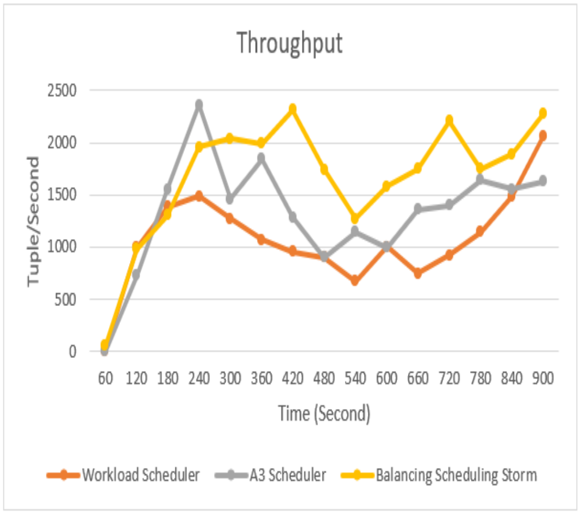

Fig. 10. Throughput Comparison Result between Three Schedulers for Spike Detection Topology.

VII. Conclusion and Future Work

In this paper, the Balance Scheduling on Storm (BSS) scheduler is proposed. It is a hybrid scheduling algorithm based on two existing scheduling algorithms: the Workload scheduler and the A3 Storm. It takes balancing the workload between the cluster nodes while minimizing the communication network between them and the inter-executor traffic. This proposed algorithm has been compared with the two existing schedulers using two metrics, throughput, and complete latency metrics. The BSS algorithm has shown better performance. The results show high throughput more than the other two algorithms and less latency. This has improved the performance of the system.

The comparison done in this paper used two essential metrics: throughput and complete latency. In future work, an enhancement in the performance will be carried out for other metrics, such as the amount of memory and resources used to give better performance and enhancement than the already given performance in this thesis. The comparison can be done with more scheduling algorithms than the two algorithms already compared with. New research could enhance the number of resources and the amount of memory and their usage instead of the balanced scheduling. The experiments would take place on more topologies to check their suitability and performance.

Acknowledgment

The authors acknowledge the Academy of Scientific Research and Technology in Egypt for its support and funding this paper within the Academy project no.: 6490.

References

[1] Y. Riahi and S. Riahi, "Big Data and Big Data Analytics: Concepts, Types and Technologies," International Journal of Research and Engineering, vol. 9, no. 5, pp. 524-528, 2018.

[2] A. C. Lyons and J. Grable, "An Introduction to Big Data," JOURNAL OF FINANCIAL SERVICE PROFESSIONALS, vol. 72, no. 5, pp. 17-20, 2018.

[3] L. Aniello, R. Baldoni and L. Querzoni, "Adaptive Online Scheduling in Storm," In DEBS 2013, 2013.

[4] A. Muhammad and M. Aleem, "A3 Storm: topology , traffic , and resource aware storm," The Journal of Supercomputing, no. 77, p. 1059–1093, 2021.

[5] R. Mall, Introduction. Real-Time Systems (Theory and Practice)., Kharagpur : Dorling kindersley (India), 2007.

[6] K. J. Giri and T. A. Lone, "Big Data - Overview and Challenges.," International Journal of Advanced Research in Computer Science and Software Engineering, pp. 525-528, 2014.

[7] G. Karypis and V. Kumar, "Multilevel k-way Partitioning Scheme for Irregular Graphs," JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING, vol. 48, no. 1, p. 96–129, 1998.

[8] L. Fischer and A. Bernstein, "Workload Scheduling in Distributed Stream Processors using Graph Partitioning," in 2015 IEEE International Conference on Big Data (IEEE BigData 2015), Santa Clara, CA, USA, 2015.

[9] A. Jain, Mastering Apache Storm, 1st ed., Packt Publishing, 2017.

[10] S. Saxena and S. Gupta, Practical Real-Time Data Processing and Analytics, Packt Publishing, 2017.

[11] B. Peng, M. Hosseini, Z. Hong, R. Farivar and R. Campbell, "R-Storm: Resource-Aware Scheduling in Storm," in ACM Middleware '15 Proceedings of the 16th ACM Annual Middleware Conference, Vancouver, Canada, 2015.

[12] J. Xu, Z. Chen, J. Tang and S. Su, "T-Storm: Traffic-aware Online Scheduling in Storm," in 2014 IEEE 34th International Conference on Distributed Computing Systems, 2014.

[13] C. Li, J. Zhang and Y. Luo, "Real-time scheduling based on optimized topology and communication traffic in distributed real-time computation platform of storm," Journal of Network and Computer Applications, vol. 87, pp. 100-115, 2017.

[14] X. Liu and R. Buyya, "D-Storm: Dynamic Resource-Efficient Scheduling of Stream Processing Applications," in 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS), Shenzhen, China, 2017.

[15] A. Muhammad, M. Aleem and M. A. Islam, "TOP-Storm: A topology-based resource-aware scheduler for Stream," Cluster Computing, no. 24, pp. 417-431, 2021.

[16] G. Karypis and V. Kumar, "METIS—A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Ordering of Sparse Matrices," University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, 1998.

[17] B. Gautam and A. Basava, "Performance prediction of data streams on high-performance architecture," Human-centric Computing and Information Sciences, vol. 9, no. 2, 2019.

[18] M. V. Bordin, A benchmark suite for distributed stream processing systems, 2017.

[19] S. Madden, "Intel Lab Data," 2004. [Online]. Available: http://db.csail.mit.edu/labdata/labdata.html. [Accessed 5 July 2021].