Cluster Computing
https://doi.org/10.1007/s10586-020-03117-y

TOP-Storm: A topology-based resource-aware scheduler for Stream Processing Engine

Asif Muhammad[1] · Muhammad Aleem[2] · Muhammad Arshad Islam[2]

Received: 10 September 2019 / Revised: 17 March 2020 / Accepted: 20 April 2020

Abstract
Like other emerging fields, Stream Processing Engines (SPEs) pose several challenges to the researchers e.g., resource awareness, dynamic configurations, heterogeneous clusters, load balancing, and topology awareness. All of these aspects play a major role in the job scheduling process. Currently, SPEs ignore topology's structure while scheduling. Due to this, frequently communicating tasks may end up at different computing nodes which causes problems for achieving the maximum throughput. In this paper, TOP-Storm—a scheduler based on topology's DAG (Directed Acyclic Graph) is proposed for Apache Storm (a popular open-source SPE) that optimize resource usage for heterogeneous clusters. The aim is to improve efficiency using resource-aware task assignments that results in enhanced throughput and optimize resource utilization. TOP-Storm is divided into two phases: In the first phase, executors are logically grouped with the help of DAG to minimize inter-group communication. In the second phase, these groups are assigned to physical nodes starting from the most powerful node. Results are generated with the help of two benchmark topologies and results are compared with two state-of-the-art scheduling algorithms. Experiment results show up to 39% and 11% improvement in throughput as compared to the default Apache Storm scheduler and R-Storm, respectively.

Keywords Stream processing engine · Topology-aware · Resource-aware · Apache storm

1 Introduction

Real-time stream processing has gained paramount attention in recent years. The real-time stream processing paradigm aims to process streaming data with low latency. Scheduling algorithms have a significant impact on latency, throughput, etc. To address the requirements of real-time stream processing, various SPEs have been proposed i.e., Apache Storm [1], S4 [2], Spark [3], Kafka [4], etc. Apache Storm is widely being utilized by the research community due to its scalability, reliability and real-time stream processing engine [5]. Over the last few years, a broad range of research work [6]– [12] has been investigated the mechanisms to improve throughput and resource utilization of Apache Storm cluster. A topology in Apache Storm defines a DAG, which represents the structure and logic of Apache Storm application. Therefore, different graph partitioning techniques were used in the literature [7, 11, 12] to place frequently communicating executors closer to each other. Readers interested in the details of the building block of Apache storm may consult [1, 13].

Apache Storm has 4 kinds of built-in schedulers. Default, Isolation, Multitenant, and Resource-Aware [14]. The default Apache Storm scheduler distributes tasks equally among computing nodes in a round-robin fashion. However, in a cluster, not all computing nodes need to belong to the same specification. This introduces a potential performance bottleneck due to an unbalanced workload distribution for computational resources. As a result, Resource-aware scheduling algorithms were proposed [9, 10, 15] to address this issue. Resource-aware scheduling considers resource availability on machines and resource requirements of workloads when scheduling jobs [16]. However, these are either static or nonadaptive. As a result,

✉ Muhammad Aleem
m.aleem@nu.edu.pk

[1] Capital University of Science and Technology, Islamabad 44000, Pakistan

[2] National University of Computer and Emerging Sciences, Islamabad 44000, Pakistan

Published online: 06 May 2020

runtime changes are not handled by these scheduling algorithms.

To address these issues, we propose TOP-Storm—a Topology-based Resource-aware scheduler for SPE (Apache Storm) to find frequently communicating executors and assign them closer to each other such that minimum nodes are utilized for topology execution. TOP-Storm is topology-aware, as it uses DAG when finding a connection between executors. It is resource-aware because physical mapping is performed based on the node's computation power. TOP-Storm finds the maximum number of executors that can be placed in a single slot (also known as worker process). After that, it reads executor's connectivity from topology's DAG. Based on this connectivity, the critical path is calculated and assigned to the most powerful machine in the cluster. This reduces communication traffic by placing highly communicating executors together and to utilize powerful machines first.

The main contributions of this work are as follows:

1. A novel scheduling algorithm TOP-Storm: A topology-based resource-aware scheduler for the heterogeneous environment;
2. We implement TOP-Storm in Apache Storm 2.0.0 [1] as an extended scheduler;
3. Performance evaluation with state-of-the-art scheduling techniques while considering two benchmark topologies. The results have shown that TOP-Storm increases average throughput by up to 39%.

The rest of the paper is organized as follows. Section 2 presents an overview of Apache Storm. Section 3 describes state-of-the-art work in this research domain. Section 4 describes the TOP-Storm research methodology. Section 5 delineates the experimental evaluation. The scheduling results and comparison to the state-of-the-art are discussed in Sect. 6. In the last section, conclusions and future work are explained.

2 Apache Storm: overview and background

Since we implemented TOP-Storm on Apache Storm [1], we briefly introduce it here. The architecture of Apache Storm follows *master* and *slave* patterns. The master and slave processes are coordinated by a third component called Zookeeper [13, 17] (Fig. 1). The architecture of Apache Storm allows only a single master node [5].

The physical architecture of Apache Storm consists of three major components. *Nimbus*, the Nimbus daemon represents the master component in the architecture of Apache Storm and distributes the work among multiple worker nodes. It assigns a task to a worker node, monitors the task progress, and reschedules the task to another worker node if a node fails. *Zookeeper* [17] coordinates and shares the information between *Nimbus* and *Supervisor* components. All states of the running tasks and the Nimbus and Supervisor daemon are stored in the Zookeeper cluster. *Supervisor*, a Supervisor is the slave or worker node, which executes the actual tasks. It initiates a new Java Virtual Machine (JVM) instance for each slot.

The Apache Storm application architecture consists of three components. *Spout* defines the source of tuples in Apache Storm. A Spout reads data from an external source and provides it to the Apache Storm topology. For example, a Spout might listen to a Twitter stream and emits this data into a stream [5]. *Bolt*, the actual processing of a task is executed by Bolts. A Bolt takes the tuples of one or multiple input streams, processes and emits them to one or multiple output streams. *Topology* defines a DAG, which represents the structure and logic of Apache Storm's real-time application. Each node in this DAG processes and forwards tuples in parallel. Each topology contains a specific configuration, which is loaded before the topology starts and does not change during runtime. A topology typically consists of Spouts and Bolts. Figure 2 shows a linear topology including one Spout and three Bolts.

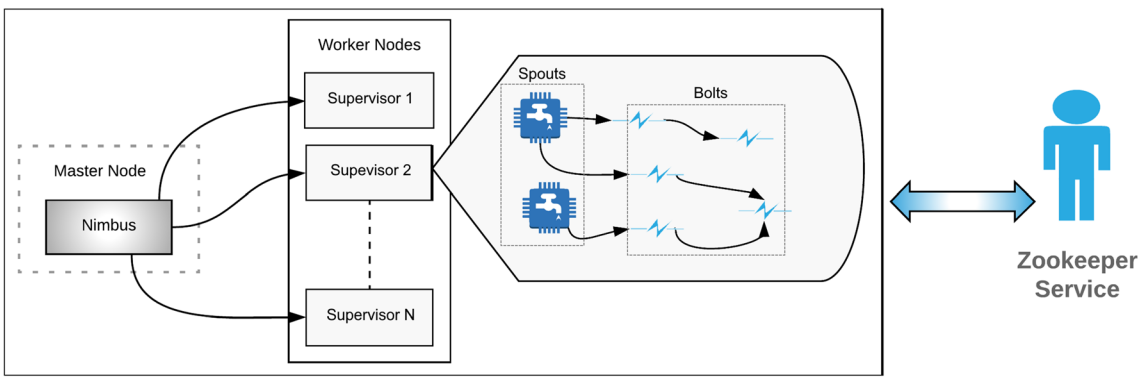


Fig. 1 The apache storm physical architecture

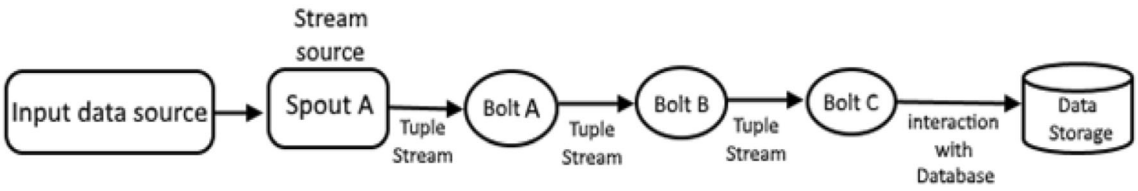


Fig. 2 Example linear topology

The default Apache Storm scheduler employs a simple round-robin strategy to produce an even number of tasks allocation. In the first step, it iterates through the topology executors and allocates them to the slots. In the next step, according to the *slot* availability, the slots are evenly assigned to the worker nodes. This scheduling produces slots with an equal number of executors and distributes slots among the worker nodes such that each worker node executes an equal number of slots.

3 Related work

The default Apache Storm scheduler is a fair scheduler that considers each node for scheduling tasks; however, it has several drawbacks. It evenly assigns workload to slots across the cluster without considering inter-node and inter-process communication, which may adversely impact the performance [6]. Several researchers have proposed scheduling algorithms to improve throughput and resource utilization of the Apache Storm cluster.

To address these issues, Eskandari et al. [12] presented an adaptive scheduling algorithm called P-Scheduler. To reduce the inter-node traffic, P-Scheduler first calculates the required number of computing nodes to execute a topology in the cluster. Afterward, it consolidates the cluster after computing the estimated load. P-Scheduler maximizes communication efficiency by placing frequently communicating pairs at a minimum distance. Despite these features, P-Scheduler has a major drawback that it is designed for homogeneous clusters only. It also assumes an equal number of slots for all the computing nodes.

The Default Apache Storm Scheduler does not consider the resource requirement of Apache Storm topologies. This can lead to resource over-utilization/under-utilization causing failure or execution inefficiency. To overcome this issue, Peng et al. [9] presented R-Storm, which implements resource-aware scheduling within Apache Storm. R-Storm is developed to increase throughput by maximizing resource utilization while minimizing network latency. When scheduling the tasks, R-Storm satisfies resource constraints and can also minimize network distance between communicating components. However, due to compile-time scheduling, R-Storm is unaware of the actual runtime workload and remains unchanged during the whole lifecycle of the application. Secondly, the scheduling is performed during compile-time only. Therefore, it is difficult to adjust the execution plan according to runtime changes.

The stream processing model works on continuous data flow. Due to this, a large amount of data arrives dynamically that cannot be evaluated in advance. The difference between batch processing and stream processing leads to a difference in scheduling and resource-allocation approaches. Smirnov et al. [18] presented a performance-aware scheduling algorithm for real-time data streaming systems. The common scheduling problem is to assign tasks to the best-fit slots, minimizing inter-node communication and maximizing CPU utilization for all nodes. With the help of a genetic algorithm, these problems are addressed by [18]. However, in this study, only linear topology is used with 1 spout and 3 bolts. Moreover, the authors have not presented the implementation details of the proposed methodology.

Data Stream Management Systems lack an intelligent system, which can adjust the scheduling plan based on resource usages. In 2017, Liu [8] presented D-Storm to address this problem. D-Storm monitors topologies at runtime to obtain their resource usages and communication pattern to avoid resource contention and to reduce inter-node communication. It makes run-time decisions to schedule tasks closely to reduce inter-node communication. It automatically re-schedules the topology whenever the resource contention is detected. Despite all these features, D-Storm has the following limitations. D-Storm is based on runtime statistics; therefore, it suffers from a cold start issue (Initially some workload/traffic load is required for this scheduler to work). D-Storm is implemented for homogeneous clusters and does not consider the network topology of processing nodes during scheduling. Similarly, D-Storm is slowest (20 times) to generate a scheduling plan as compared to the default scheduler.

The default Apache Storm scheduler distributes tasks equally among computing nodes in a round-robin fashion. However, in a cluster, not all computing nodes need to belong to the same specification. Therefore, this introduces a potential performance bottleneck due to an unbalanced workload distribution for computation resources. Yangyang Liu [7] proposed TOSS—the topology-based

scheduling algorithm to address this issue. TOSS improves performance by reducing the communication overhead. TOSS works in two phases to achieve efficiency. In the first phase, TOSS analyzes static topology structure and partitions executors in such a way that it minimizes the communication overhead between executors. In the next phase, it utilizes previously-stored run-time workload information to estimate the current workload. This phase suffers from a cold start problem because in the beginning run-time information is not available for current workload estimation. Similarly, while generating a schedule for a topology the network structure is not considered by TOSS.

Table 1 presents a comparative analysis of existing work. The table contains strengths and weaknesses of each approach. The default scheduler always allocates all the available worker nodes within a cluster, regardless of workload [19]. The over-provisioning of resources by the default Apache Storm scheduler causes low resource utilization and higher computing cost. Additionally, the application's computing requirements and communication patterns are ignored that causes delays and lower throughput [9, 10]. The default scheduler is designed for the homogeneous cluster which may introduce performance bottleneck problems [7].

4 TOP-Storm scheduler

The proposed scheduler TOP-Storm assigns topologies to a heterogeneous cluster to improve resource utilization and increases throughput as well. The TOP-Storm maps topologies by contemplating computational requirements of topologies and computation power of the node. All unassigned executors are arranged according to their communication pattern (represented by DAG). Computationally fast machines are used first ensures higher throughput. TOP-Storm consists of two phases, i.e., logical grouping and physical mapping. The logical grouping involves grouping of executors which is done using DAG. With DAG-based grouping, connected executors are put together closer to each other which reduces latency and improves throughput. The physical mapping assigns a highly communicating executors group to a node based on node's computation power (starting from most powerful to least). In this way, Resource-aware scheduling is achieved by this phase. This is an iterative process until all groups are mapped to nodes. The List of notations used in TOP-Storm's algorithm are listed in Table 2.

4.1 TOP-Storm algorithm

For efficient job execution in a topology-based system, a scheduler needs to find frequently communicating executors, which can be assigned to the same supervisor node, reducing the inter-node communication. To achieve this, TOP-Storm uses a heuristic to find such groups to minimize inter-node communication. These groups are assigned to a supervisor node with relative capacity. TOP-Storm consists of two main steps:

1. *Executor grouping*: Topology's DAG is used for executor grouping. Connected executors are placed as close as possible. This is done in Algorithm 2, which is explained below.

Table 1 Comparative analysis of existing work

	Scheduling aspects	Strength (+) / Weakness (−)
(Liu and Buyya 2018) [8]	Dynamic	+ Re-schedules when contention is spotted
	Heterogeneous	− Inefficient scheduling
	Resource-aware	
(Li and Zhang 2017) [11]	Topology-aware	+ CPU utilization-based load balancing algorithm
	Dynamic, Resource-aware	− Homogeneous solution
(Light 2017) [7]	Topology-aware	+ Communication pattern-based scheduler
	Heterogeneous	− Static scheduler
(Weng et al. 2017) [10]	Resource-aware	+ Dynamic resource-efficient scheduling
		− Topology unawareness problem
(Eskandari et al. 2016) [12]	Topology-aware	+ Graph-based adaptive scheduling
	Dynamic	− Homogeneous solution
(Peng et al. 2015) [9]	Heterogeneous	+ Compact executor assignment
	Resource-aware	− In-adaptive scheduling
(Xu et al. 2014) [19]	Dynamic	+ Heterogeneous solution
	Heterogeneous	− Cold start problem

Table 2 List of notations used in TOP-Storm's algorithm

Notations	Description
ω	Total number of slots required for topology execution
e_u	All unassigned executors for a topology
e_a	All assigned executors for a topology
e_t	Total number of executors for a topology
e_{ps}	Maximum executor per slot
e_i	Next unassigned executor
e_s	Next unassigned source executor
e_d	Next unassigned destination executor
$\propto$	Adjustment factor
s	All unassigned slots of a node
s_a	Assigned slot of the current node
s_i	A specific slot of the current node
s_n	Next free slot of the current node
$s.Sort$	Sort all slots by number of executors in descending order
n	All unassigned nodes in the cluster
n_a	An available node in the cluster
n_i	A specific node in the cluster
$n_i.GFLOPs$	Calculate computation power of i^{th} node in GFLOPs
$n.Sort$	Sort unassigned nodes by computation power in descending order

2. *Slot assignment*: Above created groups are assigned to slots. Slots are assigned starting from the most computationally powerful node. This is achieved using Algorithm 3, which is discussed below.

Algorithm 1 is responsible for generating an execution plan based on available resources and communication patterns between topology's executors. It takes unassigned topologies as input and node-wise executor assignment is the output of this algorithm. Inter-executor connectivity for all unassigned executors is retrieved (line 3). It reads the number of slots (worker process) to be used for topology execution (Algorithm 1, line 4). For logical grouping of executors to a slot, Executor Grouping (Algorithm 2) is used (line 5). Similarly, for physical mapping of a slot to a node, Slot Assignment (Algorithm 3) is used (line 6). Finally, physical mapping is handed over to Apache Storm for execution (line 7).

Algorithm 1: Main Procedure

```
1 function TOP-Storm (T);
  Input : Unassigned Topologies T
  // All unassigned topologies which need to be scheduled
  Output: Node-wise executor assignment
  // All executors of unassigned topolgies are mapped to cluster's node
2 foreach topology t_i ε T do
3 |   HashMap<source, destination> e_u = t_i.getUnassignedExecutors();
  |   // Get inter-executor connection for all unassigned executors
4 |   ω = t_i.NumWorker;
  |   // Number of worker processes for topology execution
5 |   HashMap<slot,executor> slotMap = MapExecutorToSlot(e_u,ω);
  |   // Map executors to slots
6 |   HashMap<node,slot> nodeMap = MapSlotToNode(slotMap);
  |   // Map slots to nodes
7 |   AllocateSlotsToNodes(nodeMap);
  |   // Assign slots to physical nodes
8 end
```

Algorithm 2 takes inter-executor connectivity and the required number of slots as input. As an output, it returns executor to slot assignment. It calculates the maximum number of executors to be placed in a slot (line 5). A pair of unassigned executors are selected from the top of the list and assigned to the current slot (lines 7–14). A heuristic is used to select the next executor for assignment. From the unassigned list, executor with the highest communicating frequency with already assigned executor is selected and assigned (lines 15–20). This process continues until the maximum limit per slot is reached. At the end of this process, if there are still unassigned executors, then they will be assigned in a round-robin fashion (lines 24–32).

Algorithm 2: *ExecutorAssignment*

```
1  function MapExecutorToSlot (HashMap<source, destination> e_u, ω);
   Input  : HashMap<source, destination> e_u, ω
   // Inter-executor connectivity details
   Output: HashMap<slot,executor> slotMap
   // Executor(s) to slot(s) mapping
2  s_a = 0;
3  e_a = 0;
4  e_t = e_u.Count();
5  e_ps = Ceiling (e_t / ω);
6  while e_u ≠ null do
7      e_s = e_u.getSourceExecutor();
8      e_d = e_u.getDestinationExecutor();
9      if e_s ≠ assigned && e_d ≠ assigned then
10         slotMap.add(s_a,e_s);
11         e_a++;
12         slotMap.add(s_a,e_d);
13         e_a++;
14         counter = 2;
15         while counter < e_ps do
16             e_k = e_u.getFrequentlyCommunicatingExecutor(e_s,e_d);
               // Frequently communicating executor w.r.t. source or destination
17             slotMap.add(s_a,e_k);
18             e_a++;
19             counter++;
20         end
21         s_a++;
22     end
23 end
24 if e_a < e_t then
25     counter = 1;
26     while e_a < e_t do
27         e_k = e_u.getNextExecutor();
           // Get next unassigned executor
28         slotMap.add(counter % ω,e_k);
           // Round-robin assignment
29         e_a++;
30         counter++;
31     end
32 end
```

Once executors are assigned to slots, the next step is to map slots to computing nodes. First, Algorithm 3 calculates the computation power (GFLOPs) of each node with the help of Eq. 2 (Algorithm 3, lines 4–14). Heterogenous resource-awareness is also handled by this equation. All nodes in the cluster are ranked in terms of GFLOPs irrespective of their hardware specifications. The nodes are sorted in descending order according to their computation power (lines 15–17). Similarly, slots are sorted in descending order according to the number of executors (lines 18–20). Finally, the slot with the maximum executors is assigned to the most powerful node and so on. Slots are assigned to node until all slots of that node are consumed then the next node is used (lines 21–27). In this way, minimum nodes are utilized for topology execution and inter-node communication is also reduced.

Algorithm 3 is responsible for the calculation of the computation power of each supervisor node. The computation power of each node is measured in Floating Point Operations Per Second (FLOPS) [20] that is calculated

using Eq. 1. To index each node, the FLOPS of that node is used. FLOPS are used to measure the performance of a processor while clock speed indicates the speed of a processor. The clock speed can not define the total number of calculations a processor can perform in a second. Therefore, FLOPS is a better way of knowing the throughput of a processor [21].

Algorithm 3: Slot Assignment

```
1  function MapSlotToNode (HashMap<slot,executor> slotMap);
   Input  : HashMap<slot,executor> slotMap
   // Executor(s) to slot(s) mapping
   Output: HashMap<node,slot> nodeMap
   // Slot(s) to Node(s) mapping
2  ClusterNode Nodes = ClusterInfo.getNodes();
3  Set α = 0.8;
4  foreach Node n_i ← Nodes do
5      if n_i.FreeSlots > 0 then
           // if a node has free slot(s) then calculate its computation power
6          noOfSockets = n_i.getSockets();
7          noOfCores = n_i.getCores();
8          frequency = n_i.getFrequency();
9          noOfFlop = n_i.getFlop();
10         RAM = n_i.getRAM();
11         n_i.GFLOPs = α (noOfSockets x noOfCores x frequency x noOfFlop) + (1 - α)RAM;
12         n_a++;
13     end
14 end
15 if n_a ≥ 2 then
16     n = n.Sort('Desc');
       // Sort available nodes by computation power, available slots in descending order
17 end
18 if slotMap.length() ≥ 2 then
19     slotMap = slotMap.Sort('Desc');
       // Sort mapping by number of executors in descending order
20 end
21 foreach Slot s_i ε slotMap do
22     if n_i.FreeSlots == 0 then
23         n_i = n.getNextNode();
24     end
25     nodeMap.add(s_i, n_i);
26     n_i.FreeSlots–;
27 end
```

$$ProcessingSpeed = noOfSockets \times noOfCores \times frequency \times noOfFlop; \quad (1)$$

[22], [23]

$$ComputationPower = \alpha(ProcessingSpeed) + (1-\alpha)RAM; \quad (2)$$

First, *ProcessingSpeed* is calculated using the Eq. 1 where *noOfSockets* represents the total number of sockets on the motherboard, *noOfCores* represents total cores of a processor, *frequency* represents clock frequency of a core (in Hz), and *noOfFlop* represents the total number of floating-point operations.

4.2 Analysis of adjustment factor α

In a computing node, RAM also plays an important role. Therefore, Eq. 2 is devised to give weightage for installed RAM. To accommodate the effect of RAM, we employ an adjustment factor α with a value of 0.8% for the proposed scheduling heuristic (Algorithm 3, line 11). Different α values from 0 to 1.0 are tried and finally, 0.8 is selected. With α equals 0.8, the machines with high computing power are indexed on the top. Similarly, if we set α to 0.2 then the machines are indexed w.r.t. RAM size (see Table 3). According to Table 3, Node-D represents the most powerful machine in the cluster followed by Node-E. Similarly, Node-E has a maximum of 16 GB memory whereas Node-D contains 12 GB memory. If we set α to 0.8 then the Node-D is indexed on top followed by Node-E (resulting in the desired ordering). Similarly, if we set α to 0.2 then the Node-E is indexed on top in the cluster followed by Node-D. The selected α value is valid for the employed experimental cluster. It can be different for different cluster configurations.

Table 3 Machine indexing for different α values

S #	Name	CPU (GHz)	RAM (GB)	FLOPs/cycle	# cores	CPU index ($\alpha = 0.8$)	Memory index ($\alpha = 0.2$)
1	Node-A	2.4	4	4	4	5	5
2	Node-B	2.8	8	4	4	4	4
3	Node-C	3.2	10	16	2	3	3
4	Node-D	3.4	12	16	4	1	2
5	Node-E	3.2	16	8	8	2	1

Table 4 Hardware configurations of the employed experimental cluster

Hostname	Processor	FLOPs/cycle	GFLOPs (Eq. 1)	RAM (GB)	Computation power (Eq. 2)	Index
Nimbus-Server	Intel Core i7-6700 CPU @ 3.40 GHz × 8	6816	185,395	3.7	148,317	1
Zookeeper-Server	Intel Core i7-6700 CPU @ 3.40 GHz × 8	6816	185,395	3.7	148,317	1
Node-C	Intel Core i7-6700 CPU @ 3.40 GHz × 8	6816	185,395	3.7	148,317	1
Node-A	Intel Core i7-6700 CPU @ 3.40 GHz × 4 (4 CPU Cores disabled)	6816	92,698	3.7	74,159	2
Node-B	Intel Core i5-4460 CPU @ 3.20 GHz × 4	6385	81,728	7.7	65,384	3
Node-D	Intel Core i5 CPU M 460 @ 2.53 GHz × 4	5054	51,147	3.7	40,918	4
Node-E	Intel Core i7-6700 CPU @ 3.40 GHz × 2 (6 CPU Cores disabled)	6816	46,349	3.7	37,080	5

5 Experimental details

In this section, we describe the experimental setup with software/hardware configurations. The benchmark schedulers which are used to evaluate the communication cost of each scheduler as compared to TOP-Storm. Finally, we describe applications used to evaluate the system throughput and resource usage of each scheduler.

5.1 Experimental setup

A real heterogeneous Apache Storm cluster is configured with a nimbus node, ZooKeeper node, and 5 supervisor nodes to evaluate the performance of TOP-Storm. Ubuntu 19.04 64-bit (GNOME 3.32.1) is installed on each node with network connectivity of 1000 Mb/s. Each node has Java OpenJDK 13 running on top of Ubuntu and uses Apache Storm 2.0.0[1] as a base system orchestrated by Apache Zookeeper 3.4.13, with dependent libraries zeromq 4.3.2 and JZMQ 3.1.1. Python 3.7.3 language is used to develop working scripts. The high power nodes are configured with 8 cores whereas the low power nodes have 4/2 cores. This configuration is used to achieve heterogeneity. A heterogeneous environment refers to different hardware specifications, we argue that the same hardware with different configuration (number of cores, RAM, FLOPs, etc.) can also be considered as a heterogeneous environment. TOP-Storm is built as an extension of Apache Storm 2.0.0 and the comparable approaches (default scheduler and R-Storm) are also directly extracted from this release. Table 4 contains experimental environment hardware configurations.

[1] https://github.com/apache/storm

Table 5 Different configurations used for experimentation

		Number of slots available per supervisor node		
		3	4	5
Number of slots required by the topology	3	Configuration 6	–	–
	4	Configuration 5	Configuration 4	–
	5	Configuration 3	Configuration 2	Configuration 1

5.2 Benchmark Apache Storm scheduling algorithms

TOP-Storm is compared with the following state-of-the-art Apache Storm schedulers:

1. *The default scheduler* [24–26] assigns executors to slots (worker processes) and then assigns these slots to machines in a round-robin fashion.
2. *R-Storm* [9, 27] consists of two main parts, task selection, and node selection. First, R-Storm obtains a list of unassigned tasks using breadth-first traversal. Second, a node is selected for each task based on physical distance with bandwidth consideration.

5.3 Test topology selection and evaluation metrics

Apache Storm has different types of topology structures. Among these, linear topology is one of them. The performance of TOP-Storm is evaluated with the help of two linear benchmark topologies. Word Count Topology [28–30] that breaks sentences into words and then counts the number of occurrences of each word. Exclamation Topology [31] that breaks sentences into words and then appends three exclamation marks (!!!) to the words.

For the performance evaluation, we consider the following metrics:

1. **Throughput** represents the number of tuples processed per unit time
2. **Resources used** represents the number of supervisor nodes used in topology's execution

6 Results and discussion

This section presents the evaluation of TOP-Storms using 2 topologies (listed in Sect. 5.3) with state-of-the-art algorithms (explained in Sect. 5.2). In this evaluation, we executed both topologies with 3 schedulers under different configurations. For example, the number of slots required for topology execution is varied from 3 to 5 for both topologies. Similarly, the number of available slots for each supervisor node also varied from 3 to 5 (shown in Table 5). E.g., Configuration 3 states that we have 3 slots per supervisor node and we have 5 supervisor nodes in the given cluster. As a total, we have 15 (3 slots × 5 nodes) slots available for topology scheduling. For configuration 3, 5 slots are required by topology for execution. Therefore, 5 slots from 15 available slots will be selected for topology execution by the scheduler. This selection of slots will vary from the scheduler to the scheduler based on their algorithm. Each experiment is performed for 1500 seconds while metics are recorded at a 60-seconds interval.

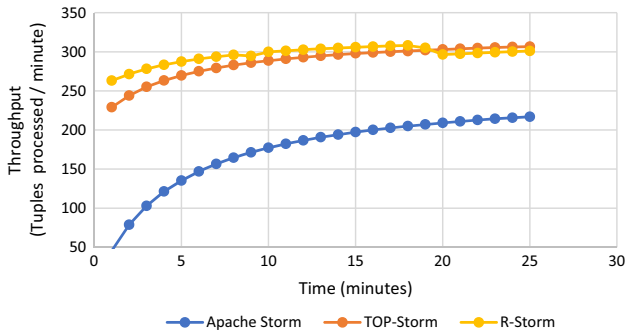


Fig. 3 Performance comparison of the scheduling algorithms for word count topology using configuration 1

6.1 Comparison with state-of-the-art schedulers

Using word count topology, performance comparison of the scheduling algorithms (Fig. 3–8), average throughput (Fig. 9), and resource usage (Fig. 10) are shown here. For configuration 1, we have 5 slots per supervisor node (total 25 slots as we have 5 supervisor nodes) and 5 slots are required by word count topology for execution. Figure 3 shows inter-executor traffic produced using configuration 1. Here, the TOP-Storm improved about 39% on average throughput as compared to the default scheduler (as shown in Fig. 9). TOP-Storm achieved these results while using only 20% computational resources (1 supervisor node only) w.r.t. default scheduler which used all 5 nodes (Fig. 10). The reason for this assignment is that TOP-Storm consumes all available slots of a selected node then moves to the next node. On the other hand, the default scheduler makes such an assignment in round-robin fashion.

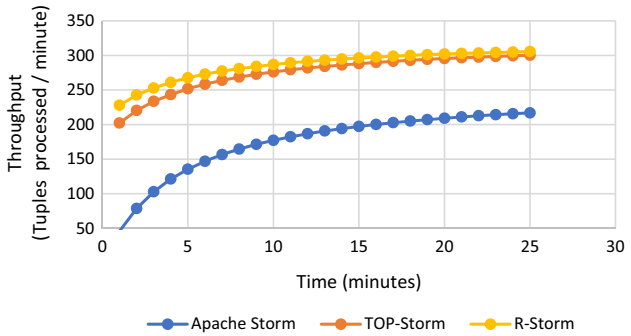


Fig. 4 Performance comparison of the scheduling algorithms for word count topology using configuration 2

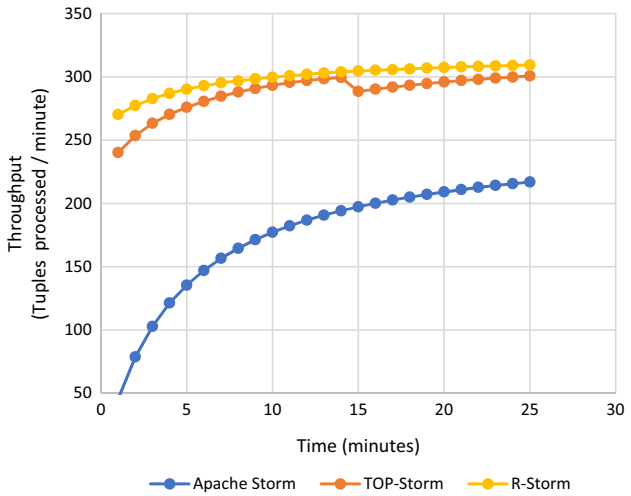


Fig. 5 Performance comparison of the scheduling algorithms for word count topology using configuration 3

The performance comparison between TOP-Storm and baseline scheduling algorithms under configuration 2 is shown in Fig. 4. Improvement in average throughput (Fig. 9) is 37% for TOP-Storm while using 60% less computational resources (Fig. 10). Figure 5 shows a performance comparison between schedulers in terms of inter-node communication. From Fig. 5, it is observed that TOP-Storm performed better than other scheduling algorithms in performance metrics. Average throughput (shown in Fig. 9) is improved to 39% for TOP-Storm using 2 supervisor nodes while the default is using 5 supervisor nodes (Fig. 10).

For the remaining 3 configurations, we do not have results for R-Storm and reason is discussed below. In configuration 4, we have 4 slots per node and 4 slots are required by wordcount topology. Figure 6 shows inter-executor traffic produced by this configuration. TOP-Storm improved about 14% on average throughput (as shown in Fig. 9). It is not a major improvement in throughput, but these results are produced with 1/5 resources as compared

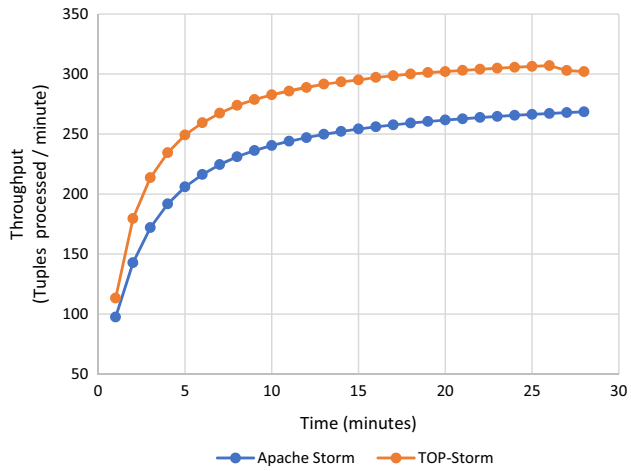


Fig. 6 Performance comparison of the scheduling algorithms for word count topology using configuration 4

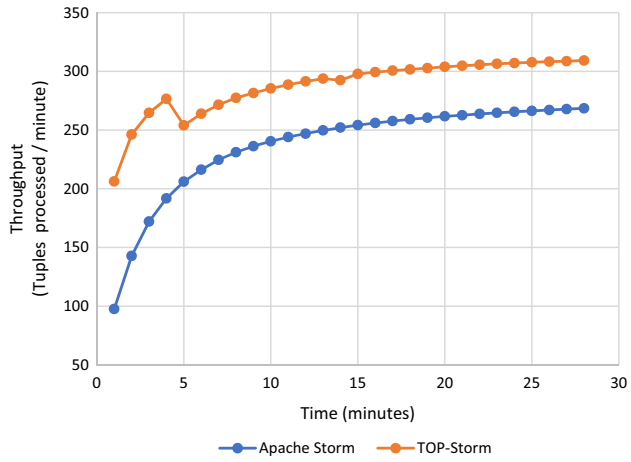


Fig. 7 Performance comparison of the scheduling algorithms for word count topology using configuration 5

to the default scheduler (Fig. 10). When configuration 5 is deployed (Fig. 7) then TOP-Storm processed 18% more tuples in the given time (Fig. 9) with 40% resources (Fig. 10). Similar experiments are performed for configuration 6 (Fig. 8) and 8% improved throughput (Fig. 9) is achieved.

The same procedure is repeated for exclamation topology using the configuration table (Table 5). Performance comparison, average throughput, and resource usage for exclamation topology are shown (Fig. 11,16,17). Figure 11 shows inter-executor traffic produced using configuration 1. TOP-Storm has 23% improvement in average throughput as compared to the default scheduler (as shown in Fig. 16). TOP-Storm achieved these results with only 20% computational resources. However, the default uses 100% available resources (Fig. 17) due to the round-robin technique.

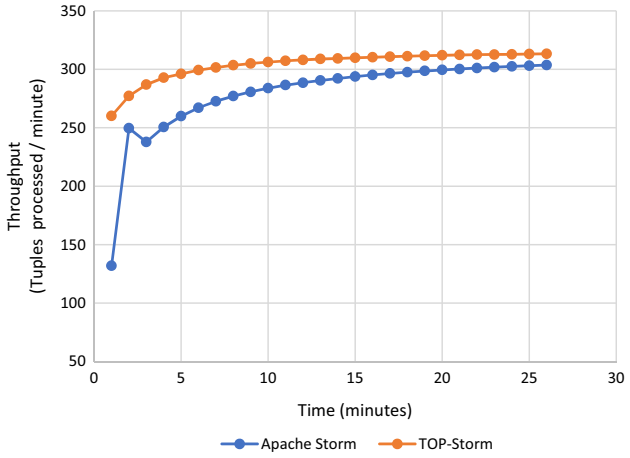


Fig. 8 Performance comparison of the scheduling algorithms for word count topology using configuration 6

The comparison of TOP-Storm with other algorithms under configuration 2 and 3 is shown in Figs. 12, 13. Improvement in average throughput (Fig. 16) is 23% with 40% resources used (Fig. 17). The reason for having a minor improvement in average throughput is that we have a total of 18 executors for exclamation topology. As the number of slots is increasing then inter-executor communication is also increasing. For configurations with fewer slots, we have better results.

Figure 14 shows inter-executor communication produced using configuration 4. TOP-Storm scheduler improved around 31% (as shown in Fig. 16) with only 1/5 resources as compared to the default scheduler (Fig. 17). When configuration 5 is deployed (Fig. 15) then TOP-Storm processed 29% more tuples in the given time (Fig. 16) with 40% resources used (Fig. 17). Similarly, Performance comparison of the scheduling algorithms for

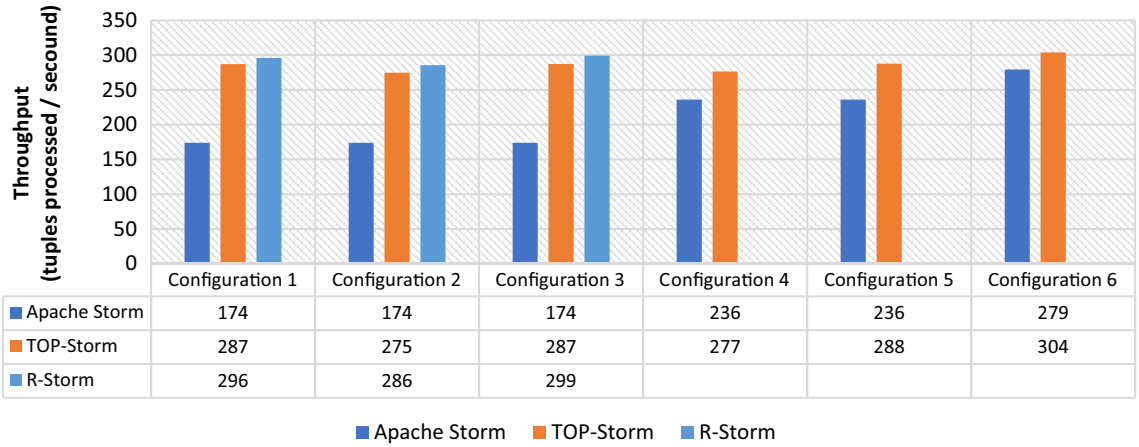


Fig. 9 Average throughput achieved for word count topology using different configurations

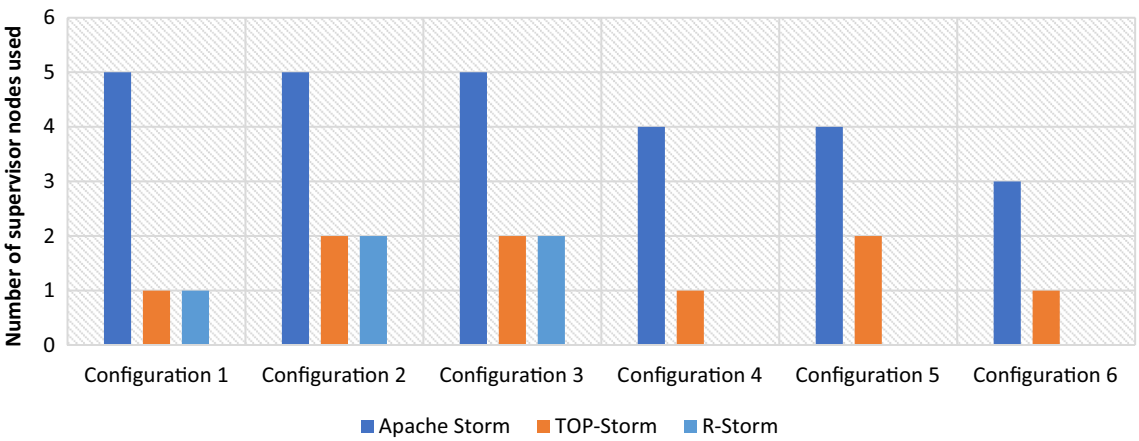


Fig. 10 The number of supervisor nodes used for word count topology using different configurations

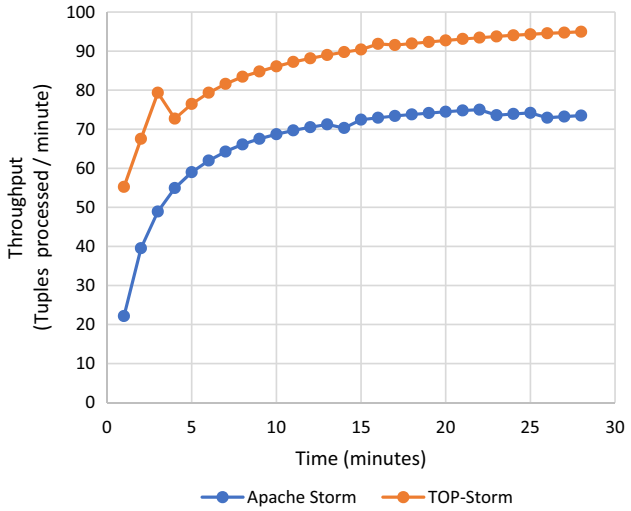


Fig. 11 Performance comparison of the scheduling algorithms for exclamation topology using configuration 1

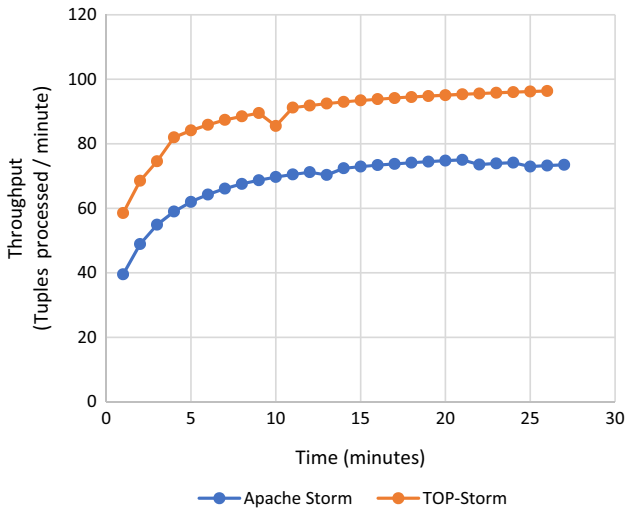


Fig. 12 Performance comparison of the scheduling algorithms for exclamation topology using configuration 2

exclamation topology using configuration 6 is shown in Fig. 18.

6.2 Discussion

From results, it is obvious that topology mapping concerning resources and communication pattern is a better scheduling choice. It performs better in terms of throughput and resource usage as compared to other scheduling algorithms. These results will be difficult to achieve if the available resources or communication patterns between executors are ignored. The default scheduler uses a round-robin approach which may increase inter-node traffic and decrease throughput. Similarly, it engages maximum

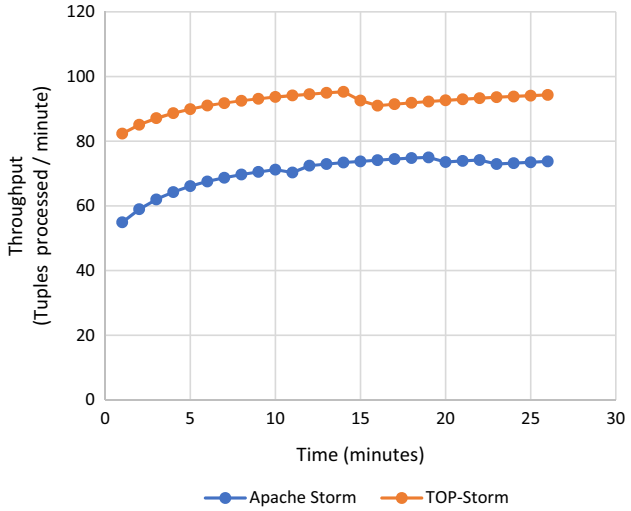


Fig. 13 Performance comparison of the scheduling algorithms for exclamation topology using configuration 3

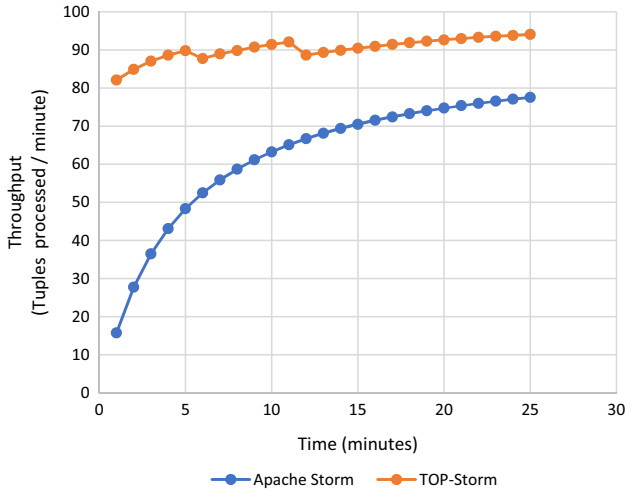


Fig. 14 Performance comparison of the scheduling algorithms for exclamation topology using configuration 4

supervisor nodes in the cluster. Conversely, TOP-Storm consumes all available slots of a node then moves to the next node if required. In this way, a compact slots assignment is achieved with a smaller number of nodes as compared to the default scheduler. This assignment reduces inter-node traffic and improves overall throughput. As a result, we have shown improvement up to 39% for word count topology in average throughput with 20%–40% resources.

R-Storm is designed to increase throughput by maximizing resource utilization while minimizing network latency. Its major limitation is that it ignores the number of required slots provided by the user at the time of topology submission. Based on resources required by topology, R-Storm distributes executors among different slots then

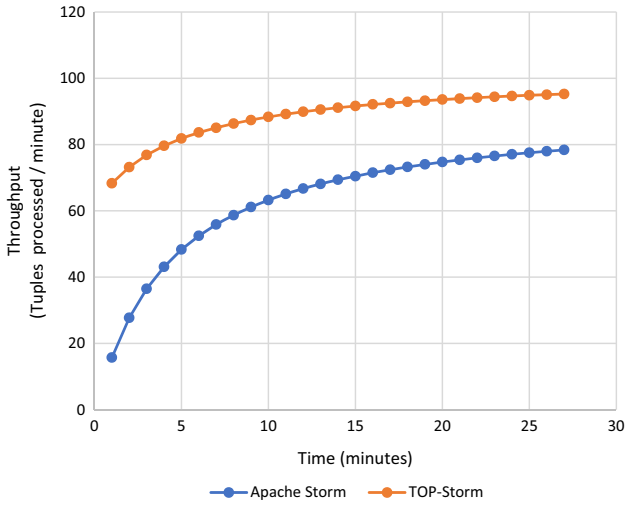


Fig. 15 Performance comparison of the scheduling algorithms for exclamation topology using configuration 5

packed slots to nodes as compact as possible. If the desired number of slots is not available, then it does not execute topology. Hence over-provisioning or under-utilization is a common scenario in this case. For example, we have 28 executors for wordcount topology. When R-Storm generates mapping then it divides them to 5 slots having 6 executors each. 2 extra executors are generated where each executor occupies 768 MB space in memory. Due to this mapping, we have R-Storm's results of configuration 1–3 (Fig. 9) only for word count topology where 5 slots are used. With over-provisioning, R-Storm performs slightly better than TOP-Storm. The same is the case for exclamation topology where we have 18 executors. R-Storm divides them into 3 slots having 6 executors each. Due to this mapping, we have R-Storm's results of configuration 6 (Fig. 16) only where 3 slots are used. In this case, when equal resources are used then TOP-Storm outperforms R-Storm by 11% on average throughput. Results showed

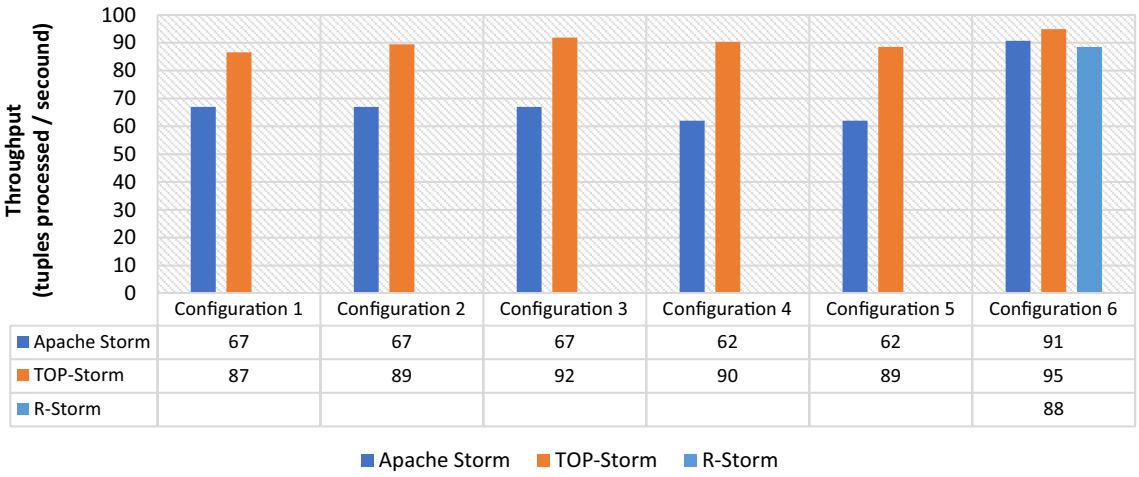


Fig. 16 Average throughput achieved for exclamation topology using different configuration

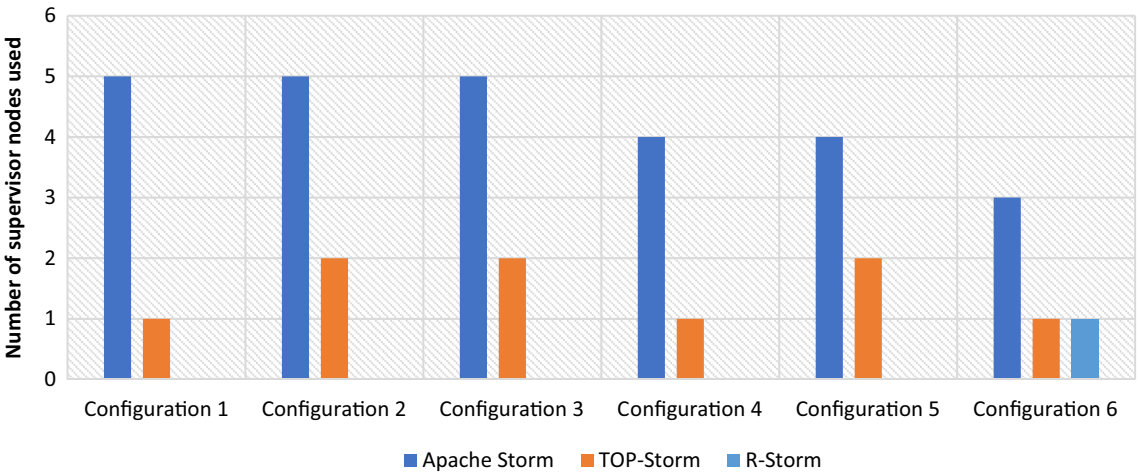


Fig. 17 Number of supervisor nodes used for exclamation topology using different configuration

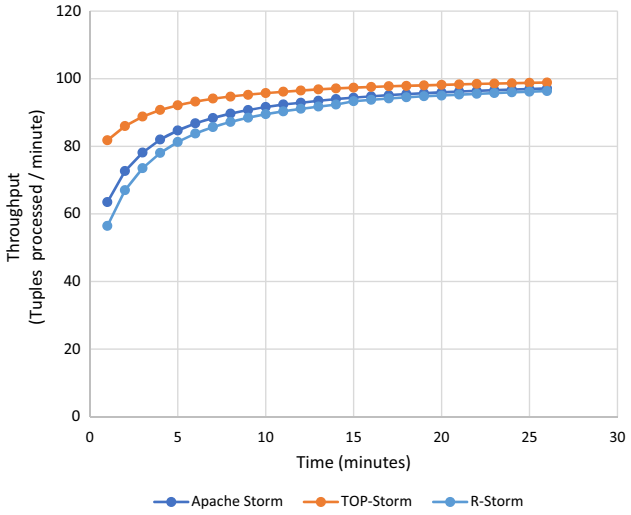


Fig. 18 Performance comparison of the scheduling algorithms for exclamation topology using configuration 6

the ability of TOP-Storm to efficiently map communicating executors closer with the help of topology's DAG, improving overall throughput.

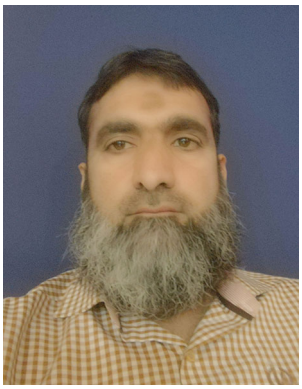
7 Conclusions and future work

In this work, we proposed TOP-Storm; a dynamic scheduling algorithm based on topology's DAG and supervisor node computing power. Firstly, we considered the topology's DAG for executors to slots grouping then, the node's computational power is used for slots to node mapping. TOP-Storm reduces resource usage as well as inter-node communication that contains overhead. We evaluated TOP-Storm using 2 benchmark topologies with 2 state-of-the-art schedulers. Results showed that TOP-Storm outperformed default scheduler in average throughput by 4%–39% with up to 80% fewer resources. For word count topology, TOP-Storm achieved the same throughput with 5% less memory usage when compared to R-Storm. Similarly, when equal resources are used then TOP-Storm outperforms R-Storm by 11% on average throughput for exclamation topology. In the future, we will include different types of topologies to check the suitability of TOP-Storm. Similarly, we have the intention to incorporate memory bound and I/O bound jobs as well. We will analyze the impact of bandwidth differences between computing nodes while scheduling a topology in a larger heterogeneous cluster.

References

1. A. S. Foundation, Apache Storm Documentation [Online]. Available: https://storm.incubator.apache.org/documentation/Home.html. (2014) Accessed 13 Nov2017
2. Apache Software Foundation, S4 Incubation Status—Apache Incubator [Online]. Available: https://incubator.apache.org/projects/s4.html. (2014) Accessed 23 Aug 2019
3. The Apache Software Foundation Apache Spark™ is a unified analytics engine for large-scale data processing, *Apache Spark*, [Online]. Available: https://spark.apache.org/. (2018) Accessed 23 Aug 2019
4. SQLstream | Streaming SQL Analytics for Kafka & Kinesis—SQLstream provides the power to create streaming Kafka & Kinesis applications with continuous SQL queries to discover, analyze and act on data in real time. [Online]. Available: https://sqlstream.com/. Accessed 06 Sep 2018
5. Illecker, M.: Real-time twitter sentiment classification based on Apache Storm (2015)
6. Aniello, L., Baldoni, R., Querzoni, L.: Adaptive online scheduling in storm, in DEBS 2013—Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (2013) pp. 207–218.
7. Light, J.: Energy usage profiling for green computing. Proceeding—IEEE Int. Conf. Comput. Commun. Autom. ICCCA (2017) vol. 2017-January, pp. 1287–1291
8. Liu, X., Buyya, R.: D-Storm: dynamic resource-efficient scheduling of stream processing applications, Proc. Int. Conf. Parallel Distrib. Syst.—ICPADS (2018) vol. 2017-December, pp. 485–492
9. Peng, B., Hosseini, M., Hong, Z., Farivar, R., Campbell, R.: R-storm: resource-aware scheduling in storm, in Middleware 2015— Proceedings of the 16th Annual Middleware Conference (2015) pp. 149–161.
10. Weng, Z., Guo, Q., Wang, C., Meng, X., He, B.: AdaStorm: resource efficient storm with adaptive configuration, in Proceedings—International Conference on Data Engineering (2017) pp. 1363–1364.
11. Li, C., Zhang, J.: Real-time scheduling based on optimized topology and communication traffic in distributed real-time computation platform of storm. J. Netw. Comput. Appl. **87**, 100–115 (Jun. 2017)
12. Eskandari, L., Huang, Z., Eyers, D.: P-scheduler: adaptive hierarchical scheduling in Apache Storm, in ACM International Conference Proceeding Series (2016) vol. 01–05-February-2016, pp. 1–10.
13. Apache Storm: Architecture - DZone Big Data. [Online]. Available: https://dzone.com/articles/apache-storm-architecture. Accessed 27 Jun 2018
14. Palmer, N.: Scheduler, in Encyclopedia of Database Systems (2016) pp. 1–1
15. Chen, M., Mao, S., Liu, Y.: Big data: a survey. Mob. Networks Appl. **19**(2), 171–209 (Apr. 2014)
16. Hussain, A., Aleem, M., Khan, A., Iqbal, M.A., Islam, M.A.: RALBA: a computation-aware load balancing scheduler for cloud computing. Cluster Comput. **21**(3), 1667–1680 (Sep. 2018)
17. Apache Zookeeper, Apache ZooKeeper— Home [Online]. Available: https://zookeeper.apache.org/. (2016) Accessed 13 Nov 2018][18] P. Smirnov, M. Melnik, and D. Nasonov, Performance-aware scheduling of streaming applications using genetic algorithm, *Procedia Comput. Sci.*, vol. 108, no. 3, pp. 2240–2249, 2017.
18. Smirnov, P., Melnik, M., Nasonov, D.: Performance-aware scheduling of streaming applications using genetic algorithm. Procedia Comput. Sci. **108**(3), 2240–2249 (2017)

19. Xu, J., Chen, Z., Tang, J., Su, S.: T-storm: traffic-aware online scheduling in storm, in Proceedings—International Conference on Distributed Computing Systems (2014) pp. 535–544.
20. FLOPS - Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/FLOPS. Accessed 30 Jan 2020
21. FLOPS (Floating Point Operations Per Second) Definition. [Online]. Available: https://techterms.com/definition/flops. Accessed 30 Jan 2020
22. Khalid, Y.N., Aleem, M., Prodan, R., Iqbal, M.A., Islam, M.A.: E-OSched: a load balancing scheduler for heterogeneous multicores. J. Supercomput. **74**(10), 5399–5431 (Oct. 2018)
23. Dolbeau, R.: Theoretical peak FLOPS per instruction set: a tutorial. J. Supercomput. **74**(3), 1341–1377 (Mar. 2018)
24. Default Scheduler, *GitHub*. [Online]. Available: https://github.com/apache/storm/blob/v2.0.0/storm-server/src/main/java/org/apache/storm/scheduler/DefaultScheduler.java. (2019) Accessed 23 Aug 2019
25. Shukla, A., Simmhan, Y.: Model-driven scheduling for distributed stream processing systems. J. Parallel Distrib. Comput. **117**, 98–114 (Jul. 2018)
26. Li, T., Xu, Z., Tang, J., Wang, Y.: Model-free control for distributed stream data processing using deep reinforcement learning. Proc. VLDB Endow. **11**(6), 705–718 (2018)
27. Resource Aware Scheduler [Online]. Available: http://storm.apache.org/releases/2.0.0/Resource_Aware_Scheduler_overview.html. (2019) Accessed 23 Aug 2019
28. Word Count, SpringerReference [Online]. Available: https://github.com/apache/storm/blob/master/examples/storm-starter/src/jvm/org/apache/storm/starter/WordCountTopology.java. (2011) Accessed 23 Aug 2019
29. Storm Topology Explained using Word Count Topology Example | CoreJavaGuru. [Online]. Available: https://www.corejavaguru.com/bigdata/storm/word-count-topology. Accessed 09 Jun 2019
30. Creating your first topology—Building Python Real-Time Applications with Storm [Book]. [Online]. Available: https://www.oreilly.com/library/view/building-python-real-time/9781784392857/ch03s03.html. Accessed 05 Sep 2019
31. Exclamation Topology, GitHub [Online]. Available: https://github.com/apache/storm/blob/master/examples/storm-starter/src/jvm/org/apache/storm/starter/ExclamationTopology.java. (2019) Accessed 23 Aug 2019



Asif Muhammad received the M.S. degree in Software Engineering from Shaheed Zulfikar Ali Bhutto Institute of Science and Technology (SZABIST), Islamabad, Pakistan. His research interests are big data processing, web mining, scheduling mechanism for heterogeneous computing, and performance analysis. He is currently a Ph.D. scholar at Capital University of Science and Technology, Islamabad, Pakistan.

Muhammad Aleem received the Ph.D. degree in computer science from the Leopold-Franzens-University, Innsbruck, Austria in 2012. His research interests include parallel and distributed computing comprise programming environments, multi-/many-core computing, performance analysis, cloud computing, and big-data processing. Formerly, he was serving as assistant professor at Capital University of Science and Technology Islamabad. He is currently working as associate professor at National University of Computer and Emerging Sciences, Islamabad, Pakistan.

Muhammad Arshad Islam completed his doctorate from University of Konstanz, Germany in 2011. His dissertation is related to routing issues in opportunistic network. His current research interests are related to MANETs, DTNs, social-aware routing and graph algorithms. He is currently working as associate professor at National University of Computer and Emerging Sciences, Islamabad, Pakistan.