Journal of Big Data

RESEARCH

Open Access

A scheduling algorithm to maximize storm throughput in heterogeneous cluster

Hamid Nasiri[1*], Saeed Nasehi[1], Arman Divband[1] and Maziar Goudarzi[1]

*Correspondence:
hnasehiri@ce.sharif.edu

[1] Department of Science, Sharif University of Technology, Tehran, Iran

Abstract

In the most popular distributed stream processing frameworks (DSPFs), programs are modeled as a directed acyclic graph. Using this model, a DSPF can benefit from the parallelism capabilities of distributed clusters. Choosing a reasonable number of vertices for each operator and mapping the vertices to the appropriate processing resources significantly affect the overall system performance. Due to the simplicity of the current DSPF schedulers, these frameworks perform poorly on large-scale clusters. In this paper, we present a heterogeneity-aware scheduling algorithm that finds the proper number of the vertices of an application graph and maps them to the most suitable cluster node. We begin with a pre-processing step which allocates the vertices to the given cluster nodes using profiling data. Then, we gradually increase the topology input rate in order to scale up the application graph. Finally, using a CPU utilization model which predicts the CPU workload based on the input rate to vertices and the processing node's CPU characteristics, we identify the bottlenecked vertices and allocate new instances derived from them to the least utilized processing resource. Our experimental results on Storm Micro-Benchmark show that (1) the prediction model estimate CPU utilization with 92% accuracy. (2) Compared to the default scheduler of Storm, our scheduler provides 7 to 44% throughput enhancement. (3) The proposed method can find the solution within 4% (worst case) of the optimal scheduler, which obtains the best scheduling scenario using an exhaustive search over problem design space.

Keywords: Stream processing, Scheduling, Heterogeneous, Throughput, Parallelism

Introduction

In the stream processing applications such as network monitoring, online machine learning, fraud detection, signal processing, and sensor-based monitoring, the infinite data sequence must be processed uninterruptedly. Apache Storm [1] is among the most popular distributed stream processing frameworks for such applications. This framework models applications as a directed acyclic graph (DAG), called *topology*. A topology in Storm can be executed after determining the structure of the execution topology graph and the number of instances (tasks) used for each application component. Afterward, the Storm scheduler maps all topology tasks to processing elements (*PE*s). By

default, Storm assigns tasks according to the Round-Robin algorithm regardless of the processing power of each PE [2].

Based on the computation requirements and the input rate of an application in stream processing, sufficient processing resources must be taken to prevent CPU overloading. The required processing power remains steady in applications with constant input rates, like surveillance applications, weather analysis, and predictive maintenance. So there would be an excellent opportunity to optimize the mapping of the processing resources to these applications to gain the highest possible throughput, especially when the processing resources are heterogeneous.

In heterogeneous clusters, an optimal selection of the number of instances for each component and their assignment to PEs play a significant role in achieving high throughput and efficient utilization of resources. While application-to-architecture mapping is an old classic design automation problem, the features below make Storm optimization distinctive:

1 The user determines the number of instances of each component, and can be set based on the executing cluster.
2 The computation load of each instance is not steady and can be changed by tuning the input tuple rate.
3 The input rate of each instance determines the workload of its downstream instance(s).

Several scheduling methods [3–6], are proposed to improve the naive Storm scheduler, but most focus on resource management in homogeneous clusters. However, none of these works consider the heterogeneity of cluster nodes. So they perform poorly on heterogeneous clusters, the most prevalent of which are in real-world data centers. The authors of [7–10] also present methods to dynamically scale Storm clusters in response to processing demand or input data rate. For both scheduling and scaling solutions, a primary user graph is submitted to Storm for execution, and the user determines the number of instances for each component.

In this paper, we propose a novel scheduling algorithm, which (1) calculates the near-optimal number of instances for each component to create a fitting topology for a particular heterogeneous cluster; and (2) decides which machine is most suitable for each task based on its processing characteristics. The proposed algorithm tries to fully utilize the processing resources for a given heterogeneous cluster while preventing over-utilization. In order to prevent overutilization, a heterogeneity-aware model is proposed to predict the CPU utilization of each task running on a specific machine. Figure 1 shows the architecture overview of Apache Storm with the proposed heterogeneous scheduler on a cluster consisting of n worker nodes. The scheduler uses the user topology graph and profiling data as inputs to create a graph of execution topology. Then it maps the output graph to the worker nodes regarding their capability.

Different types of topologies from Micro-Benchmark [6] and Storm-Benchmark [15] are executed on a heterogeneous cluster to evaluate the efficiency of the proposed algorithm. To the best of our knowledge, no similar scheduling algorithm produces the execution graph to maximize the throughput of an application running over a

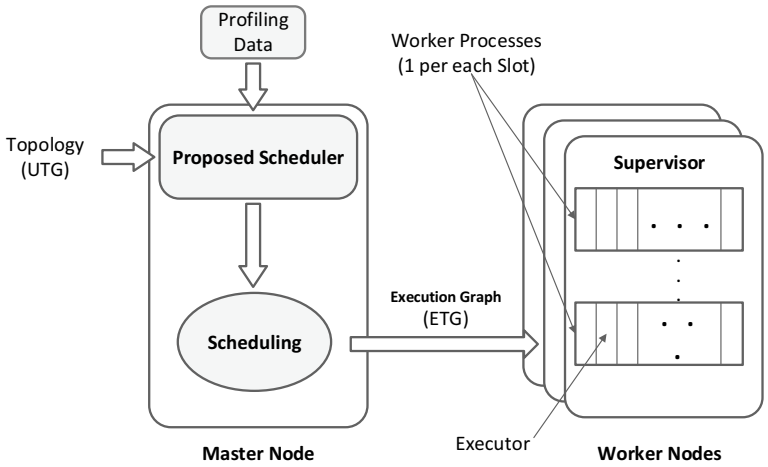

Fig. 1 Architecture overview of Apache Storm with proposed scheduler

heterogeneous cluster, So we compare our work with the optimum scheduler. The optimum scheduler obtains the best execution graph using an exhaustive search over the problem design space. The experimental results show that our algorithm provides 7 to 44% throughput enhancement in comparison with the default scheduler of Storm. Moreover, real occupied CPU utilizations show that the proposed method utilizes all processing resources efficiently. Comparing them with the predicted values demonstrates that our prediction model can estimate CPU utilization with over 92% accuracy.

We make the following major contributions to this paper:

1 First, we present a prediction model to anticipate the CPU utilization considering the CPU's characteristics and input rate
2 Then, using the above model, we provide a novel heterogeneity-aware scheduling algorithm for Apache Storm to fit a topology graph to a specific heterogeneous cluster to maximize the general-purpose stream processing applications throughput
3 Finally, we implement the proposed algorithm in Apache Storm and evaluate it using topologies from Micro-Benchmark [6] and Storm-Benchmark [15] for both throughput and resource usage efficiency.

The remainder of this paper is organized as follows. First, preliminary information about stream processing and the Apache Storm architecture is provided. Then, the motivation for considering resource heterogeneity, the problem definition, and the details of the proposed algorithm are presented in sections "Motivation", "Problem definition", and "Proposed Algorithm", respectively, followed by the evaluation of the proposed prediction model and scheduling algorithm in section 6.

Background and related works

Stream processing

Big data processing can be divided into two main categories: Batch processing and Stream processing. In Batch or off-line processing, data is prepared before processing. In stream or online processing, a possibly infinite sequence of data items is generated

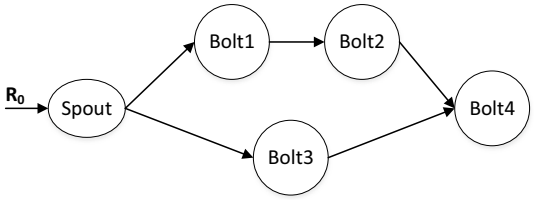

Fig. 2 A sample user graph with 1 spout and 4 bolts

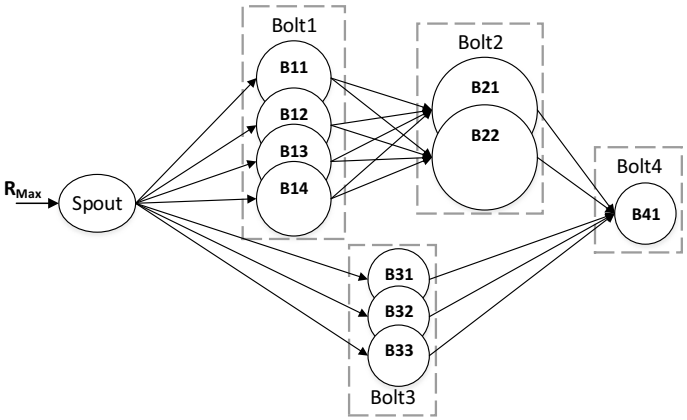

Fig. 3 An execution graph for Fig. 2 with different number of instances for each bolt

continuously in time and must be processed in real-time. Apache Hadoop is one of the most popular frameworks for batch processing, which uses *MapReduce* programming model. There are several large-scale computing architectures customized for batch processing of big data applications [11]; however, they are not suitable for stream data processing because, in the MapReduce paradigm, all input data needs to be stored on a distributed file system like HDFS, before starting to process. Many distributed frameworks have emerged to address large-scale stream processing problems; however, Apache Storm is one of the best-known stream processing framework for low latency and high throughput applications, among them all [13, 29].

Apache storm programming model

Apache Storm is an open-source, real-time, distributed processing platform that processes the unbounded data stream. In Storm, each program is modeled as a directed acyclic graph (DAG) called *topology*. A topology consists of two types of components. The component responsible for input stream production is *spout* and the component in charge of data processing is *bolt*. This primary topology is known as the user topology graph (*UTG*) [12]. Figure 2 depicts a user graph with one spout and four bolts.

Each component of the user topology graph may have several instances (*tasks*) that determine its parallelism degree. All component instances execute the same operation on different parts of the upstream data. A topology with distinct parallelism degrees, forms the execution topology graph (*ETG*) [12]. Figure 3 illustrates an example execution graph corresponding to the user graph in Fig. 2. In this figure, the parallelism

degrees of Bolt1, Bolt2, Bolt3, and Bolt4 are 4, 2, 3, and 1, respectively. We assumed each bolt of the topology has a variable size because each bolt occupies different CPU utilization, depending on its input rate and computation requirements. The structure of the execution topology graph dramatically affects overall throughput and average tuple processing time, two major stream processing considerations.

Once a topology is assigned to processing elements (Fig. 4 shows a sample assignment of execution topology graph Fig. 3), the spout(s) brings data into the system and sends them to the bolt(s). As processing progresses, one or more bolts may write data out to a database or file system, send a message to another bolt (s), or make the computation results available to the user [12].

Apache storm execution architecture

A Storm cluster consists of two kind of nodes: *master* node and *worker* nodes. The master node runs a daemon called *Nimbus*, which is the central component of Apache Storm. The main responsibility of the nimbus is distributing tasks on the worker nodes. Worker nodes do the actual execution of the topology. Each worker node (machine) runs a daemon called *Supervisor*, which executes a portion of the topology. The configuration of worker node determines how many *worker processes* (correspond to *slots*) it provides for the cluster. Each worker process runs a Java Virtual Machine (JVM), in which several threads (known as *executor*) are running.

A scheduling solution specifies how to assign tasks to worker nodes. Apache Storm's default scheduler uses a simple Round-Robin method to map executors to worker processes. Then, it evenly distributes worker processes to available slots on worker nodes. Figure 4 shows how Storm default scheduler distributes all tasks of the execution topology graph of Fig. 3 among four machines of a heterogeneous cluster.

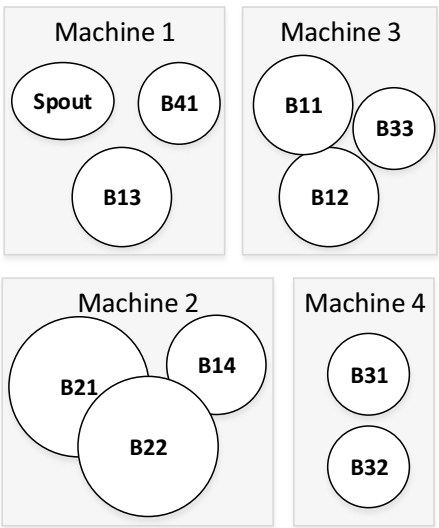

Fig. 4 Task assignment of Storm's default scheduler for execution graph Fig. 3

Literature review

Many scheduling algorithms are proposed to optimize throughput or resource utilization in Apache Storm [17–21]. One of the primary considerations in task assignment is communication cost; Aniello et al. proposed two schedulers for Storm [3]. Both schedulers find hot edges of topology and put related tasks alongside each other, but the difference is in how they determine hot edges. In R-Storm [6], B. Peng et al.. have presented a more intelligent scheduling algorithm. This algorithm is resource-aware and classifies resource constraints into soft and hard. R-Storm considers the CPU usage of homogenous machines and cannot be applied in heterogeneous environments because it uses the same unit for CPU utilization of different machines. Rychl et al. [10] presented a heterogeneity-aware scheduler for the stream processing frameworks. However, the main idea of this scheduler is based on design-time knowledge as well as benchmarking techniques. As a drawback, this scheduler makes the scheduling decision by trial and error and gives no information about the optimum number of instances.

Other papers deal with dynamic elasticity; chiefly [7, 9, 22, 23] change the scalability by tyning the parallelism degree of graph nodes. They monitor the data transfer rate of graph edges to determine their required parallelism. In these papers, the total number of available machines is unlimited. B. Lohrmann et al. [8] focused on latency constraints and presented an elastic runtime strategy that monitors and measures necessary metrics to make proper scale-up or scale-down operations. V. D. Veen et al. [24] design and implement elasticity at the virtual machine level. They monitor several metrics such as processor load, the size of an input queue, and the number of tuples emitted and acknowledge to decide whether to increase the number of virtual machines or decrease it. Stela [9] presents an on-demand mechanism for scale-out or scale-in operation, depending on user request. However, in the Stela, parallelism can be changed dynamically, but the user defines the total number of tasks. D-Storm [25] models the scheduling problem as a bin-packing problem and proposes a First Fit Decreasing (FFD) heuristic algorithm to solve it. As a goal, they try to minimize inter-node communication; by dynamic rescheduling and do not consider the overhead of reassignment of the tasks. I-Scheduler [26] is another scheduling algorithm that tries to overcome the complexity of the task-assignment problem by reducing the total number of tasks. It finds highly communicating tasks by exploiting graph partitioning techniques and takes only one instance for each of them. In this way, it reduces the total number of tasks and overall communication cost.

In both scheduling and scaling solutions, a primary execution graph is submitted to Storm to run; the user determines the number of instances for each component of this graph. In the case of Storm, the number of instances is defined before running the topology, and we have to restart the entire system to change that. Some algorithms like Stela [9], take a specific number as the maximum number of instances and change the number of instances running on one executer to scale up or down at run-time. In comparison, our algorithm offers a fitting execution graph for a cluster of heterogeneous machines. In the case of changing the number of machines, it can be re-executed to create a new execution graph.

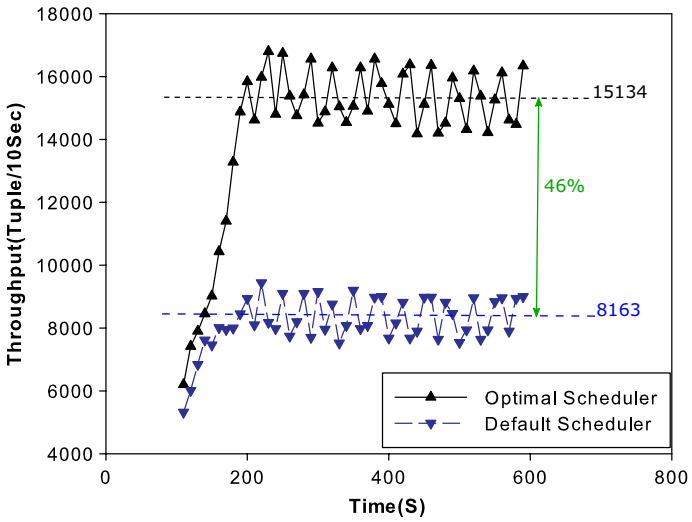

Fig. 5 Throughput comparison of default and optimal schedulers for Linear topology

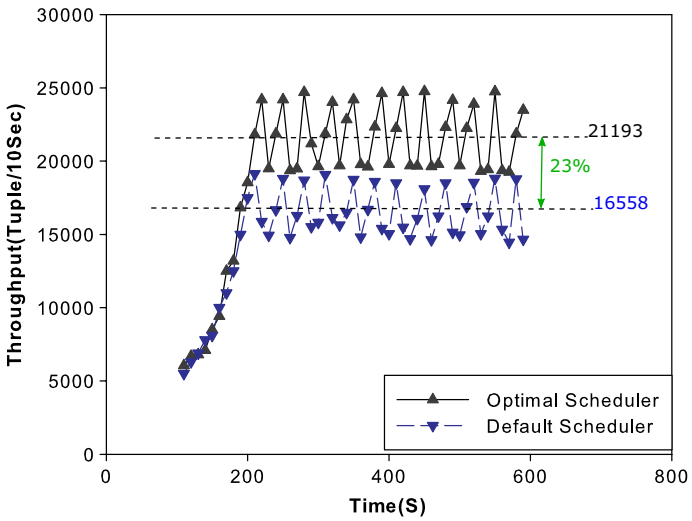

Fig. 6 Throughput comparison of default and optimal schedulers for Diamond topology

Motivation

In the last decade, CPUs have become faster and more powerful, mainly by increasing the frequency of their clocks or by developing new architectures. As a data center grows over time, it would have several generations of processors, making it a heterogeneous processing environment. Most big data applications today are processed on large-scale computer systems, such as data centers. In practice, we have heterogeneous clusters as the infrastructure for big data processing [30, 33].

However, Apache Storm does not consider resource heterogeneity in its scheduling framework, despite the prevalence of heterogeneous clusters in existing data centers. Due to this, its schedulers perform the same task assignments in homogeneous and

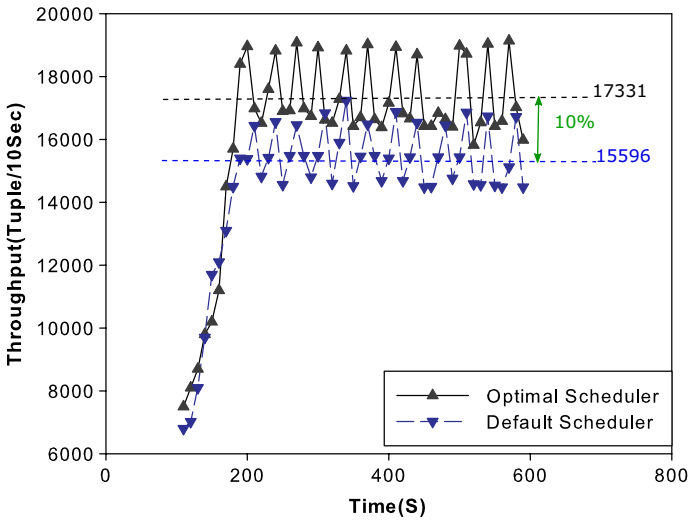

Fig. 7 Throughput comparison of default and optimal schedulers for Star topology

heterogeneous systems. To demonstrate the inefficiency of Storm's default scheduler in heterogeneous environments, we ran three basic topologies from the Storm Micro-Benchmark [6] on a cluster of three different machines (the experimental setup is described in "Evaluation" Section). Figures 5, 6 and 7 represent the total throughput of running intended typologies when their tasks are assigned to the cluster nodes by the Storm default and optimal scheduler.

The optimal scheduler is a brute-force algorithm that performs an exhaustive search on the task-assignment design space to find the best placement for all topology tasks on the target cluster's machines. Using this algorithm, the overall topology throughput is calculated for all possible placements, and the placement that results in the maximum overall topology throughput is selected. According to Figs. 5, 6 and 7, the scheduling scenario significantly impacts the overall throughput, and there is a significant gap between the current and the maximum achievable throughput. Consequently, at first glance, we may prefer the optimal scheduler over the default Storm scheduler to obtain higher throughput and efficiency. However, the optimal scheduler has a much longer execution time.

To determine the time complexity of the optimal scheduling algorithm, we have to limit the total number of topology tasks. Assuming each machine can execute at most k_j tasks simultaneously, the total number of tasks is limited to the processing power of machines. Hence, all possible non-negative integer solutions for task assignment can be obtained using the following equation:

$$\begin{aligned} & x_1 + x_2 + x_3 + ... + x_n \leq \sum_{j=1}^{m} k_j \\ & s.t. \quad \forall i = 1...n \quad x_i \geq 1 \end{aligned} \tag{1}$$

In this equation, m is the number of candidate machines for task placement, k_j is the maximum number of tasks that can be executed on jth machine simultaneously, n

is the total number of components, and x_i is the total number of tasks for ith component. By solving this equation, the time complexity of optimal scheduling algorithm is obtained $O(c(\sum_{j=1}^{m} k_j, n))$. For example, running the optimal scheduler for mapping a topology with four bolts ($n = 4$) on three machines ($m = 3$) with same processing power ($\forall j = 1, 2, 3 \quad k_j = 10$) checks *27,405* possibilities and lasts about *18* hours, by a server with four Xeon 5560 processors and 8 GB of memory. When we have a heterogeneous cluster in which the machines have different processing powers, this time is increased exponentially.

The problem of scheduling in heterogeneous systems must therefore be addressed by a new scheduler that assigns tasks faster than the optimal scheduler and results in higher throughput than the default scheduler. The following section describes the problem definition, followed by a description of our scheduling method in depth in "Proposed algorithm" section.

Problem definition

We are trying to find an appropriate number of tasks and then assign them to the most appropriate machines to maximize the overall throughput. Every component has a distinct processing requirement, which determines the number of instances to be taken from it. Furthermore, in a heterogeneous cluster, the number of instances for each component is strongly influenced by the type of machine to which they are assigned. Based on these processing requirements and the processing power of machines, how can we create enough tasks and schedule them so that the cluster is utilized fully and overall throughput is maximized, but no machine is exceeding its processing capabilities? We will try to answer this question. Table 1 summarizes the parameters used in our approach.

Table 1 Terms and definitions of the approach.

Symbols	Descriptions
N_{T_i}	Number of available machines with type T_i
N_{C_i}	Total number of instances of component i
m	Total number of worker nodes (machines)
n	Number of topology's components
PT_{iw}	Processing throughput of task i running on machine w
MAC_w	Available CPU capacity of machine w
TCU_{ij}	Occupied CPU utilization by ith task on machine j
e_{ij}	Average tuple execution time of task i on machine j
MET_{ij}	Miscellaneous execution time of Storm for task i running on machine j
R_0	Topology initial input rate
α_i	Tuple division ratio of ith component
IR_i	Input rate of ith task
PR_i	Processing rate of ith task
OR_i	Output rate of ith task

Problem statement

In the context of our system, each machine (worker node) has one worker process, which has a particular processor architecture and a specific processing budget. It is assumed that each task in the execution topology graph is assigned one executor, so the total number of tasks equals the total number of executors.

Let $TC = (T_1, T_2, T_3, ...)$ be the set of all types of machines in the target cluster, and N_{T_i} shows the number of available machines with type T_i, on that cluster. For a heterogeneous cluster with T types of machines, the summation of N_{T_i}s ($i = 1...T$) shows the total number of machines (m). Also let $UTG = (c_1, c_2, c_3, ...)$ be the set of all components within a user topology graph, and similarly, N_{C_i} says how many instances each component has. Here the summation of N_{C_i}s shows the total number of tasks (n).

Problem formulation

Our aim is to maximize throughput by finding the near-optimal number of instances for each component and arranging them appropriately on a cluster of heterogeneous machines while avoiding scheduling that would overutilize the CPUs of those machines. In Storm, a topology's overall throughput is calculated as the sum of all tasks' processing throughputs [6]. Since each component may have several instances, for each component, we sum the processing throughput of its instances (PT_{iw}s) and then these values are added together to obtain the overall throughput of a topology. To ensure that no machine is over-utilized, we put a constraint on *Machine Available CPU capacity* (MAC_w). For each machine, this constraint checks that the summation of CPU utilization of its tasks (TCU_s) does not exceed its CPU's total utilization limit. In the beginning, the value of (MAC)s is assumed to be 100. Additionally, all components must have at least one instance to create the minimal execution topology graph. Hence, the objective of our problem is as follows:

$$\begin{aligned} &\textit{maximize} && \sum_{j=1}^{n} \sum_{k=1}^{N_{C_j}} \sum_{w=1}^{m} PT_{iw} \\ &\textit{subject to} && \forall w = 1...m \quad MAC_w \geq 0 \\ &&& \forall j = 1...n \quad N_{C_j} \geq 1 \end{aligned} \tag{2}$$

where

$$i = \left(\sum_{l=1}^{j} N_{C_l} \right) + k \tag{3}$$

$$PT_{iw} = \begin{cases} PR_i & \text{If task } i \text{ is running on machine } w, \\ 0 & \text{Otherwise} \end{cases} \tag{4}$$

For each task i, we run it on machine w ($w = 1...m$) and use equation (3) to make unique indexes. By doing so, we can access the processing throughput for each task within the implemented Storm system 1.

If tasks are assigned incorrectly, or numbers of instances are taken inappropriately for components, the CPUs on physical machines may be overused or underused. However, adjusting these two options to obtain a maximum throughput without machine overloading results in more state space and computation complexity. Considering each task as an item that has specific profit (PT_{iw}) and weight (TCU_{iw}) and each machine as a knapsack with a specific capacity (MAC_w), our problem can be mapped to the *multiple Knapsack optimization* problems [28]. Due to the NP-hardness of the classic Knapsack problem, finding an optimal solution to this problem is computationally impractical.

Several meta-heuristic algorithms exist to obtain the near-optimal answer to the *multiple Knapsacks problem*, but most require considerable computation time. Therefore, fast scheduling is essential in the stream data processing. The tuple overloading state is, for example, caused by a slow scheduler in the event of a machine failure. Furthermore, any reconfiguration in the cluster requires a new task assignment, so a lazy scheduler is unacceptable. In order to solve the problem mentioned above, we propose a new heuristic algorithm characterized by both a short execution time and high throughput.

Proposed algorithm

Algorithm overview

Considering $UTG = (c_1, c_2, c_3, ...)$ as a user topology graph, we gradually scale it up over a given cluster by increasing the topology input rate and taking more instances from the compute-intensive components. This process continues until all cluster nodes have been utilized. As Fig. 8 represents, our scheduling algorithm has two phases: initialization and iteration (separated by dotted lines). In the initial phase, we take a user topology graph and a set of e_{ij}s (profiling data), then take one instance from each component; finally, we map this primary execution graph to the cluster using *FirstAssignment* procedure. In the iteration phase, we employ two options, including increasing topology input rate and taking new instances. By iteratively using these options, the algorithm progressively increases throughput and fully utilizes all available machines.

At each iteration, *MaximizeThroughput* procedure updates the *MAC* values for all machines and check if any machine is over-utilized. When there are no over-utilized machines, the topology input rate is increased until at least one machine is over-utilized. In this situation, the algorithm takes a new instance from the component, which one of its instances is the bottleneck. The algorithm then attempts to assign the instance to the most appropriate machine. If there is enough capacity to map the new instance, it is added to the execution topology graph (*ETG*). The input rate increment must be reduced if no candidate machine can be identified. The process repeats until the input rate increment reaches its threshold; therefore, there is no more capacity left on the cluster. Lastly, the iterations are complete, and the execution topology graph and mapping are obtained.

As part of our algorithm, we need to know the amount of CPU usage on all possible types of machines before mapping an instance to the correct machine. For better estimation of these values, we propose new metrics and present a formula that uses profiling data to predict the *TCU* of the intended instance on each machine. We describe this formula in detail in the following section.

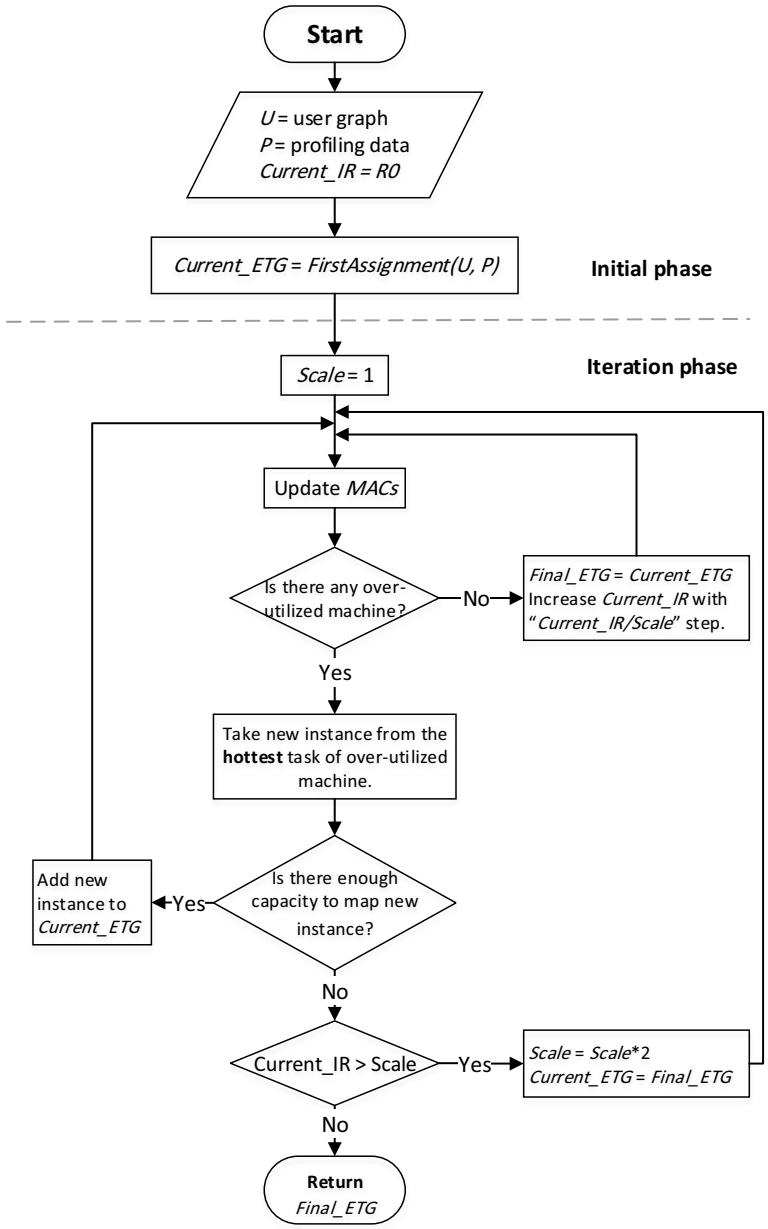

Fig. 8 Proposed scheduling algorithm overview

CPU usage prediction

There have been many studies that have attempted to classify and predict CPU utilization patterns of devices and servers in data centers, as well as traffic patterns on networks, to predict resource exhaustion [34–38]. The goal of these studies is to develop predictive models that can identify when a device or server is likely to run out of resources, such as CPU, memory, or disk space, or when a network link is likely to become congested. As we focus on the CPU intensive streaming applications, we do not consider the network and memory behaviors and it make the CPU usage prediction problem to a simpler one

and hence it can be used in our algorithm with low time complexity while according our experiments it has enough accuracy in for our problem.

According to our experiments, for each task, the value of *TCU* depends on its input rate, computation complexity, and the type of machine used to perform the task. Based on our observations, we assume a task's CPU usage linearly increases in terms of input rate increment. So we use (5) to estimate the *TCU* of ith task on jth machine as follow:

$$TCU_{ij} = (e_{ij} \times IR_i) + MET_{ij} \tag{5}$$

In this formula:

- e_{ij}: Is the average execution time of the ith task on the jth machine to process a single tuple of data. In other words, it is the time taken by a single task to process a single unit of input data.
- IR_i: Is the tuple input rate of the ith task that determines its workload.
- MET_{ij}: Is the miscellaneous execution time of Apache Storm for the ith task on the jth machine. It represents the fixed overhead time taken by Storm for a given task on a particular machine, irrespective of the input rate. This overhead time includes the time taken by Storm to set up and tear down the task, perform network communication, and perform other miscellaneous operations required to execute the task. As we mentioned, this value is considered a constant for a given task and machine combination.

According to our experiments, we observed that for each task i running on machine j, the values of e_{ij} and MET_{ij} are independent of the input rate IR_i. This means that these two variables can be treated as constants for a given task and machine combination. Based on this observation, we can use these constants to estimate the CPU usage of the task i on machine j for a given input rate IR_i.

By using this formula, we can estimate the execution time of the task i on machine j for different input rates IR_i. However, this formula assumes that e_{ij} and MET_{ij} remain constant for a given task and machine combination, which may not be true in all cases.

Our algorithm is designed based on grasping the execution characteristics of different tasks. We use pre-process profiling to know the performance characteristics of the individual types of nodes employed in the cluster for different tasks [27]. In profiling, each task is run on all types of machines, and each time we increase its input rate while it reaches the maximum value of its CPU usage. At this point, we measure the CPU usage, its corresponding input rate, and the value of e_{ij} which can be measured using *get_execute_ms_avg()* Java function or Storm's UI. Knowing *TCU*, *IR* and e_{ij}, the value of MET_{ij} is calculated using (5). The reasoning behind choosing the maximum point is that the variation in measured CPU utilization is very low when the processing load is either relatively low or relatively high [14]. As the number of arrival tuples increases, the portion of the miscellaneous overhead of Storm and CPU idle time are decreased; thus, the prediction of CPU usage at the maximum point is quite accurate. After profiling, we have a constant e_{ij} and MET_{ij} for all possible assignments of topology's components to cluster's nodes. Thus, we can predict its *TCU* for each arbitrary value of *IR*.

As topology input rate has a domino effect on all tasks, we define a new metric (called α) to estimate the output rate of each task. For task i, α_i is a given parameter (extracted from profiling data) that shows the average ratio of its output tuples count to its input tuples count. Now we can calculate the input rate of each downstream bolt using equation 6 and then obtain its corresponding CPU usage by formula 5.

$$IR_{next\_stage\_task} = \sum_{i=1}^{y} \left(\frac{OR_i}{x} \right) \tag{6}$$

where

$$OR_i = IR_i \times \alpha_i$$

In this equation, x is the number of downstream tasks which are fed by ith task, and y is the number of tasks that are feeding the intended downstream task *(next stage task)*. Here, OR_i is output rate of ith task, and IR_i is its input rate.

First assignment

To meet the first constraint of our problem, we use the *FirstAssignment* procedure to take one instance from each component of the given topology. This procedure takes a user topology graph *UTG* and a set of e_{ij}s and MET_{ij} (profiling data) as input, according to Algorithm 1. For a given topology with an initial input rate R_0, it estimates the input rate of each component, using α values and equation 6. Then, for each component, it predicts the *TCU* on different machines using profiling data and formula 5 and assigns the component to the machine, which results in the least *TCU*. In the case of a successful assignment, one instance of the component is added to the temporary execution topology (*Current_ETG*). After mapping all the components, a primary execution topology with a specified task assignment is given to the *MaximizeThroughput* procedure.

Algorithm 1: First Assignment

Input : User graph U, Profiling information P
Output: Assigned user graph $Current\_ETG$ to the target cluster

```
Current_IR = R0
Current_ETG = Null
foreach component i in U do
    Using P map i on the machine which results the least TCU
    Add i to Current_ETG
end
```

Maximize throughput

In the second phase, we use a progressive algorithm to efficiently use all processing power of the cluster nodes and maximize the overall throughput. To do that, another procedure called *MaximizeThroughput* takes profiling data and a primary execution topology graph from the *FirstAssignment* procedure as inputs. After a limited number of iterations, this procedure results in a final execution topology graph (*Final_ETG*) in which it determines how many instances each component has, where each instance must be placed on the target cluster, and the maximum value of topology input rate which cluster can tolerate.

Algorithm 2 represents pseudo-code of *MaximizeThroughput* procedure. Here, variable *Scale* is defined to control the amount of input rate increment, which its value is initialized to be 1. At the beginning of the algorithm, we assume *Current_ETG* is mapped to the cluster, and its input rate starts from R_0. In the next step, the algorithm increases the input rate progressively, and at each iteration, it updates the value of *MAC* for all machines using formula 5. After that, it checks if any machine is over-utilized. Now there are two possibilities:

1- No machine is over-utilized; in this situation current execution topology graph (*Current_ETG*) and its corresponding input rate are retained in *Final_ETG* as the latest stable state; and then topology input rate is increased by a factor proportional to *Currect_IR/Scale*. By incrementing the topology input rate, all tasks' processing load will be increased. Thus their utilization on the corresponding machine needs to be updated, so we return to line 1 to evaluate new conditions. 2- At least one machine is over-utilized; in this situation, we take a new instance from the component corresponding to the task with the highest CPU usage (*hottest* task) in the first over-utilized machine. This new instance should be assigned to one of the available machines, so we look for the most suitable machine to map this instance on. When there is at least one machine with enough capacity to serve this new instance, it is added to the *Current_ETG* alongside its assignment information; then, we go to line 1 to update the *MAC*s. However, if no candidate machine is found to serve that, the termination condition is checked. By taking a new instance from the ith component, this instance will process a portion of the upstream data (IR_i). Hence, the input rate of other instances, including the *hottest* task, is decreased, so over-utilization is solved.

While current input rate (*Current_IR*) is greater than value of *Scale*: I) The amount of input rate increment will be decreased by a factor proportional to $\frac{1}{2}$, in other words the variable *Scale* is duplicated. II) The latest stable state of scheduling is recovered by copying the *Final_ETG* to the *Current_ETG*, because the current rate exceeds the cluster processing capacity. III) Return to line 1 to update the value of *MAC*s.

Algorithm 2: Maximize Throughput

```
   Input  : User graph Current_ETG, Profiling information P
   Output: Final_ET with its mapping information on target cluster
 1 Update MACs using CPU prediction formula and P if no CPU over-utilization then
 2 |   Final_ETG = Current_ETG
 3 |   Current_IR += Current_IR/Scale
 4 |   go to line 1
 5 else
 6 |   Take new instance h from hottest task
 7 |   if enough capacity exists on cluster then
 8 |   |   Add h to Current_ETG
 9 |   |   go to line 1
10 |   else
11 |   |   if Current_IR > Scale then
12 |   |   |   Scale = 2 * Scale
13 |   |   |   Current_ETG = Final_ETG
14 |   |   |   go to line 1
15 |   |   else
16 |   |   |   return Final_ETG
17 |   |   end
18 |   end
19 end
```

In this procedure, we regulate the incrementation of input rate in lines 3 and 12; and use take new instance option in lines 6–8. This process is repeated until the termination condition ($Current\_IR \leq Scale$) is satisfied; it means the input rate cannot be increased anymore, and no capacity is left in the cluster to place any new instance. Now all machines are almost fully utilized, and the final execution graph and its corresponding assignment are created, so the algorithm ends. The computational complexity of this algorithm depends on the total capacity of the cluster and the TCU s, so the worst-case computation complexity of this algorithm can be calculated as $log(TAC_m / \sum_{i \in T} min(TCU_{iw}), \forall_{w \in m})$ where the TAC_m stands for the total capacity of the all machines in the cluster and TCU_{iw} is the occupied CPU utilization of the task i on machine w.

Evaluation

Experimental setup

To evaluate the proposed scheduler, we have implemented it in Storm as a new scheduler [31]. Using this scheduler, we run benchmark topologies on a cluster of four heterogeneous machines. Table 2 shows the specifications of cluster nodes. In our cluster,

Table 2 System characteristics.

Cluster Node	Memory	Processor	Network adapter	Operating system
Machine 1	2GB	Pentium Dual-Core 2.6 GHz	1 Gb/s	Ubuntu 16.04
Machine 2	4 GB	Intel Core i3 2.9 GHz	1 Gb/s	Ubuntu 16.04
Machine 3	6 GB	Intel Core i5 2.5 GHz	1 Gb/s	Ubuntu 16.04
Machine 4	4 GB	Intel Core i3 2.9 GHz	1 Gb/s	Ubuntu 16.04

Table 3 Profiling of topologies' tasks on cluster's machines.

Task Type	Machine 1	Machine 2	Machine 3
lowCompute	0.0581 (s)	0.107 (s)	0.0916 (s)
midCompute	0.103 (s)	0.1844 (s)	0.168 (s)
highCompute	0.1915 (s)	0.3449 (s)	0.3207 (s)

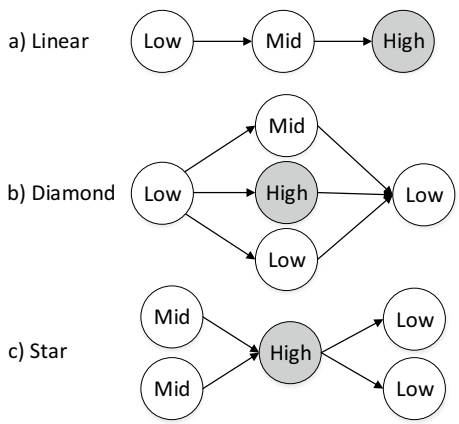

Fig. 9 Layout of Micro-benchmark [6] topologies

all machines are connected through a switch with 1 Gb/s network adapters. One of the Core i3 machines is used as a master node, which runs Zookeeper and Nimbus daemons, and other machines are configured as worker nodes. All experiments are obtained with Apache Storm 0.9.5 installed on top of Ubuntu 16.04.

However, Storm topologies can be an arbitrary DAG; most topologies are a mixture of three basic topologies: Linear, Diamond, and Star (Fig. 9). *Linear* topology has one source, a sequence of intermediate components, and a sink. When there are several parallel components between the source and sink, it is known as *Diamond* topology, and a *Star* topology has multiple sources connected to a single component. This intermediate component is the parent of multiple sinks. There are some production applications from industry like *PageLoad* topology, *Processing topology*, and *Network Monitoring topology* [7], which has been used in [6, 7, 9] to evaluate their works. All these topologies are a combination of Linear, Diamond, and Star topologies and are good choices where the processing power, network capacity, and topology connectivity must be considered. In this work, we only focus on the required processing power by each component and have no network consideration; therefore, we used three basic topologies from Micro-Benchmark [6] to evaluate the effectiveness of our scheduling algorithm. All topologies in this benchmark are made from three types of CPU intensive components called *lowCompute*, *midCompute*, and *highCompute*. Table 3 shows the profiling data e_{ij} of each component, on all types of machines which mentioned in Table 2.

To maximize the system's throughput, the algorithm tries to make a fitting execution topology graph for a specific heterogeneous cluster. Since finding the optimum number of instances for each component has a huge design space, we used two simple topologies from Storm-Benchmark [15]. They both contain two components and make it possible to verify how well the algorithm calculates the number of instances for each component. As there is no similar scheduling algorithm to produce the execution graph according to the computing power of heterogeneous machines, we compared its results with the optimal execution graph.

Experimental evaluation

In this section, we evaluate our proposed method in four directions. At first, we evaluate our proposed CPU usage predictor module. After that, we evaluate the efficiency of our proposed method in the execution graph creation. And finally, since our goal is to maximize the total throughput of the cluster, we consider the total throughput and the utilization of the machines as our next evaluation metrics. We have two baselines to compare the results with. The first baseline is Apache Storm's default scheduler. As we mentioned before, it maps executors to worker processes using a Round-Robin method and then, the worker processes are evenly distributed to slots on worker nodes. In order to find the best possible solution, an exhaustive search over problem design space is used as the second baseline.

Evaluation of CPU usage formula: The first part of our proposed algorithm, which needs to be verified, is the CPU usage prediction formula. An interesting outcome of the empirical case studies is that the difference between measured *TCU* and its calculated value is very low whenever the input rate is either relatively low or relatively high. Thus, the prediction of CPU usage is quite accurate when the CPU is either lightly or heavily

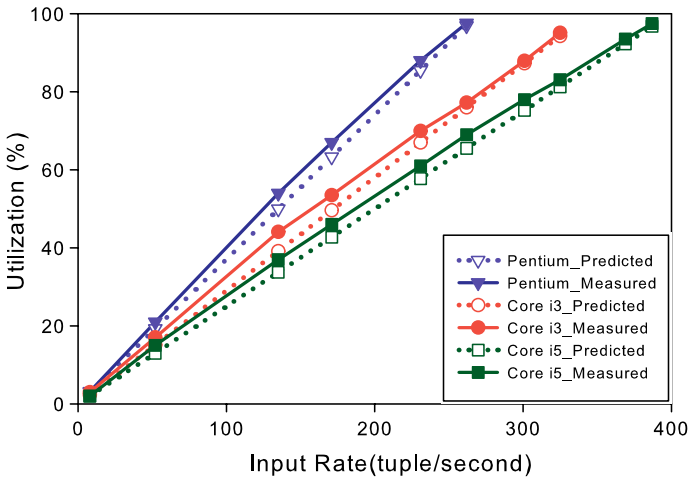

Fig. 10 Predicted and measured CPU utilization comparison for highCompute bolt on different machines in Linear topology

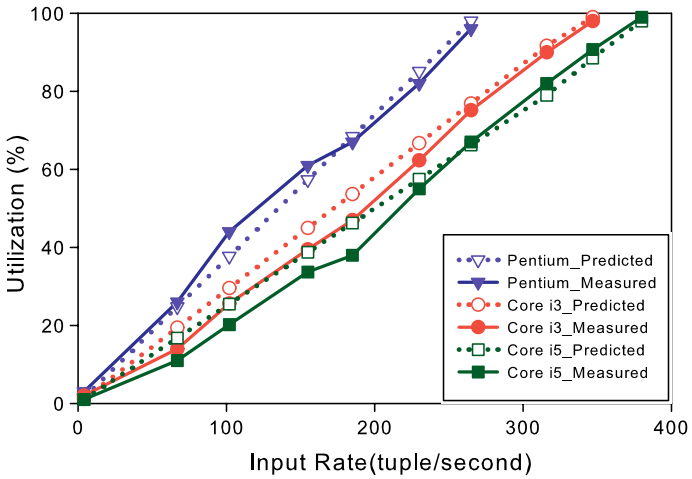

Fig. 11 Predicted and measured CPU utilization comparison for highCompute bolt on different machines in Diamond topology

loaded. Our experiments show that when the CPU usage is moderate, the measured *TCU* has more variation than its calculated value and thus is less predictable. However, even the largest difference between measured and predicted *TCU* always was less than 8%.

Our experiments in this part cover different conditions for a unique bolt, including all possible combinations of three types of processors and three different structures of topologies. For each experiment, the *highCompute* bolt (gray bolt in Fig. 9) is placed on a single machine, and its upstream bolts are placed on the powerful enough machines such that they can utilize it fully. For example, in the first experiment *highCompute* bolt of Linear topology is assigned to the machine with a Pentium processor, and its actual CPU usage is measured when the input rate of topology is eight tuples per second. Then the input rate is increased by a factor of a random number between 20 and 80, and new

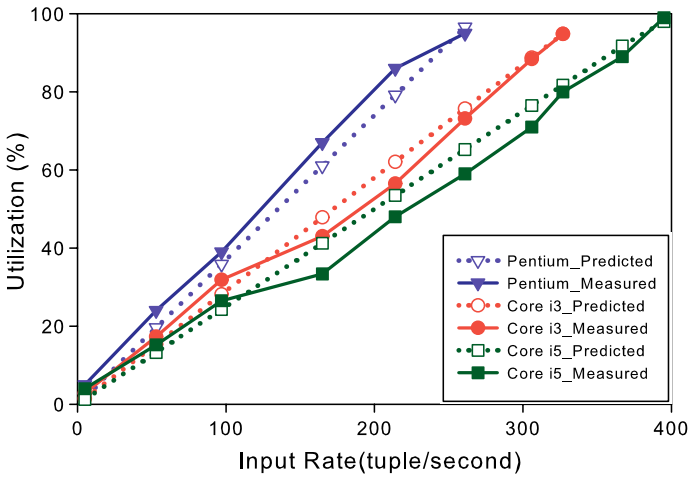

Fig. 12 Predicted and measured CPU utilization comparison for highCompute bolt on different machines in Star topology

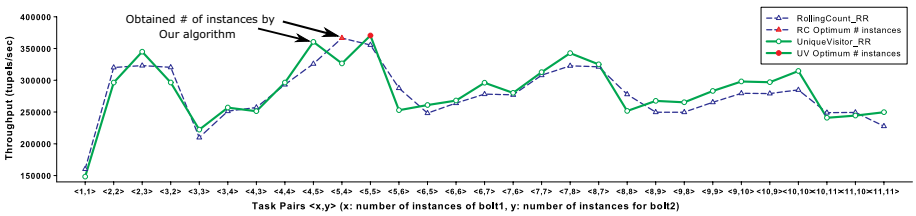

Fig. 13 Maximum achievable overall throughput in terms of number of instances

CPU usage is measured again. This process is continued while the processor becomes over-utilized, so the input rate cannot be increased anymore.

Figures 10, 11 and 12 represent the measured and predicted *TCU* of *highCompute* bolt for different values of input rate on three different types of CPUs for Linear, Diamond, and Star topologies, respectively. Here the unfilled dotted lines show the predicted value of *TCU* obtained by formula 5 and the filled dotted lines show real *TCU*, measured by the collector tool [16].

Evaluation of execution graph: One outcome of our scheduling algorithm is obtaining the near-optimal number of instances for each component. Figure 13 depicts overall throughput by different number of instances for both *RollingCount* and *UniqueVisitor* topologies from Storm-Benchmark [15]. The <*x,y*> pairs on the horizontal axis shows the number of instances for bolt1 and bolt2, respectively, in both topologies. To observe the effect of the structure of the execution graph on overall throughput, different execution graphs are scheduled using the default scheduler of Storm. Then our algorithm is performed to determine how well it calculates the number of instances for each topology. Our algorithm obtained pair <*5,4*> for *RollingCount* topology, which exactly is the optimal number of instances. It also obtained pair <*4,5*> for *UniqueVisitor* topology, which has the closest number of instances to the optimal pair <*5,5*>, and it eventuates only 2% throughput decrement than optimal throughput. The obtained pairs by our algorithm for both topologies are shown by the arrow in Fig. 13.

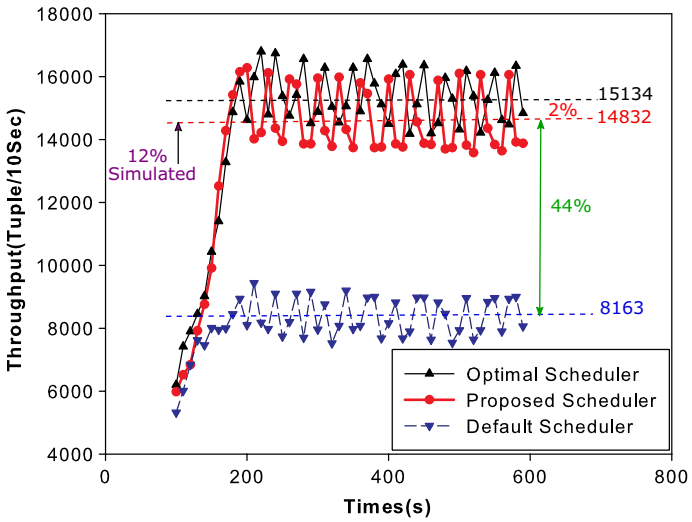

Fig. 14 Experimental throughput comparison of default, proposed and optimal schedulers for Linear topology

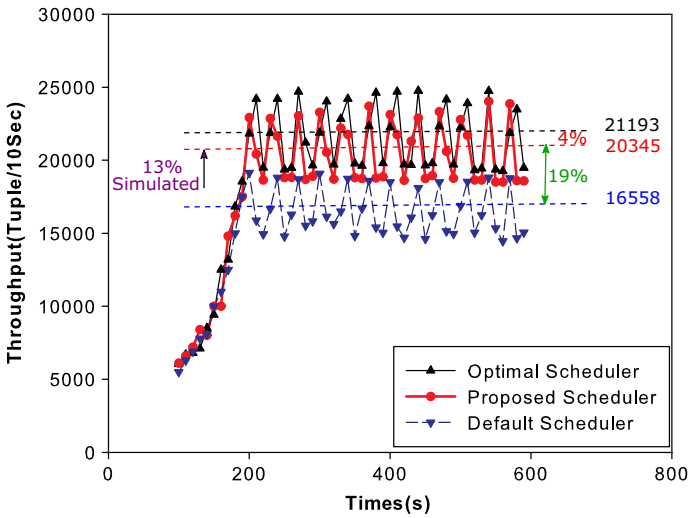

Fig. 15 Experimental throughput comparison of default, proposed and optimal schedulers for Diamond topology

Throughput Comparison: We evaluate the performance of the proposed algorithm in term of throughput. Here the efficiency of its task assignment is compared with Storm's default and optimal scheduler. Figures 14, 15 and 16 depict the experimental results of our scheduler in comparison with other schedulers. According to this figure, our scheduling method provides 7% to 44% throughput enhancement compared with Storm's default scheduler, while it can find the solution within 4% of the throughput of the optimal scheduler in the worst case (Fig. 15).

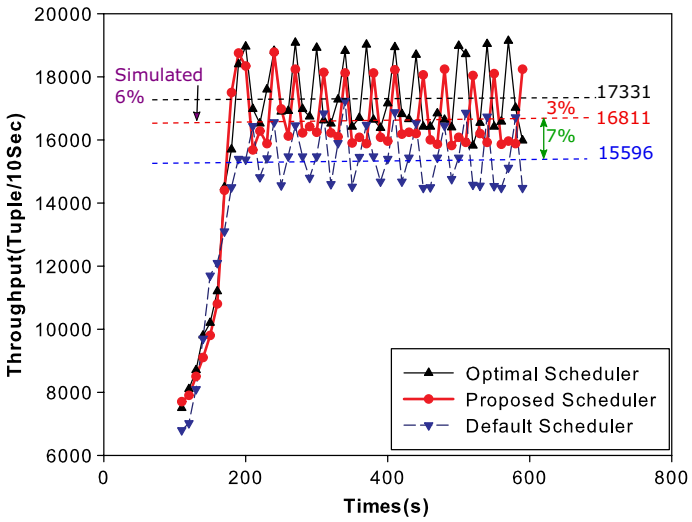

Fig. 16 Experimental throughput comparison of default, proposed and optimal schedulers for Star topology

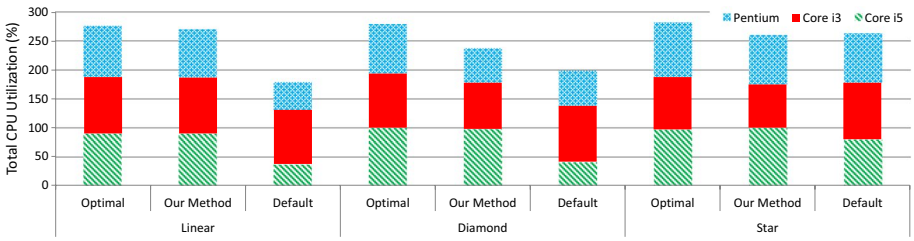

Fig. 17 Obtained CPU utilizations of worker nodes by different scheduler for Micro-benchmark topologies

As we can see, for Linear topology, our scheduler results in 44% higher throughput than the default scheduler, while it is 2% less than optimal throughput, but for other topologies, it provides less gain. This is due to the different characteristics of the target topologies. In some topologies such as Linear, the main challenge is the processing power, so the proposed scheduler would result in almost the best achievable throughput, but when other resources such as network or memory get bottleneck, we need to consider several factors to obtain a better result.

Utilization Comparison: Normally better resource utilization results in higher throughput, but to make sure that our scheduler has efficient CPU usage, we need to compare CPU utilization of all schedulers. In Fig. 17 we can see the CPU utilization of cluster nodes for Linear, Diamond, and Star topologies under different scheduling methods. In all cases, the optimal scheduler uses the CPUs efficiently, so it has the highest summation of CPU utilization percentages.

When Diamond and Linear topologies run on the cluster, our scheduler has more CPU usage and, therefore, higher throughput than the default scheduler. However, in the case of Star topology, its total CPU usage is less than the default scheduler,

although it still has higher throughput. Our scheduler uses the most powerful processor (Core i5) better than the default scheduler. Therefore, it always uses the processing resources more efficiently than the default scheduler.

Large-scale clusters evaluation

System Simulation: Due to resource limitation and real requirements of big data applications we need to use a modeling scenario. Generally, simulation makes it possible to model bigger distributed stream processing systems. To simulate the real use case scenario in which Apache Storm is being used, we needed to simulate real heterogeneous environments. Our simulation scenario takes the processing characteristics of both cluster nodes and topology components for each specific scheduling policy. Then it reports the overall throughput of existing machines' topology and CPU utilization [32].

To ensure that our simulator has enough accuracy, we performed it in the same conditions as our real experiments. As shown in Figs. 14, 15 and 16 the difference between implementation and simulation results is less than 13% in the worst case, which means the simulator has acceptable accuracy, so we can use this simulator to evaluate our scheduler in the case of large-scale clusters.

Simulation Results: For each topology (Linear, Diamond, and Star), we need to find an appropriate execution graph according to the specification of cluster nodes. Therefore, we first run our algorithm to determine the number of instances for each component for the intended cluster. Now we can fairly compare only the effectiveness of scheduling policies (proposed and default schedulers) in terms of overall throughput and resource utilization. These simulations are executed for three different clusters, with the combinations of heterogeneous machines, shown in Table 4.

Our obtained simulation results report both throughput and utilization per node of the intended cluster. We calculate the overall throughput of topology by adding together these throughput values. For utilization, we calculate a weighted average of reported utilization, with weights determined by Eq. 7. Because of cluster heterogeneity, it is vital to give more value to machines with more processing capacity. Therefore, first, we determine the weight of each machine using profiling data according to Eq. 8, then calculate overall utilization using Eq. 7.

Table 4 Number of machines and component's instances of considered scenarios.

Scenario	Cluster type	# of M1	# of M2	# of M3	Tasks								
					Linear			Diamond			Star		
					L	M	H	L	M	H	L	M	H
1	Small	2	2	2	3	3	7	12	5	7	9	10	5
2	Medium	10	10	10	41	42	35	67	15	46	45	34	25
3	Large	20	70	90	201	156	341	397	208	91	327	156	206

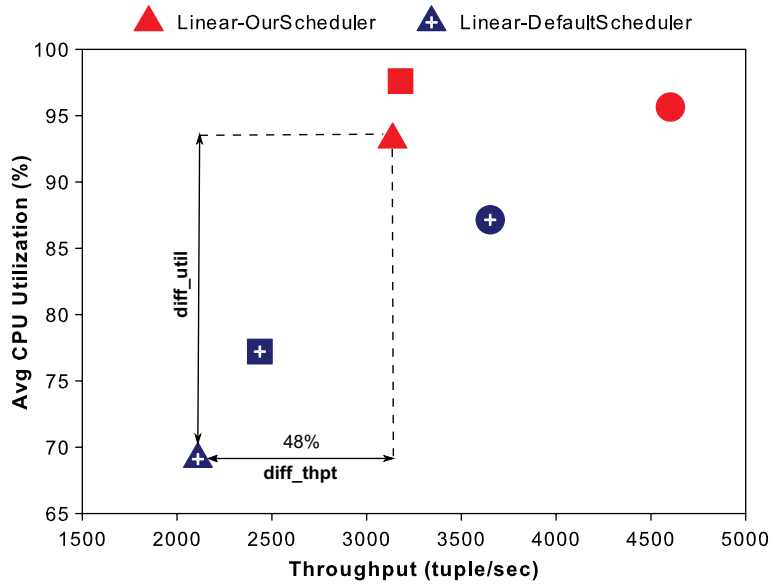

Fig. 18 Simulation results comparison of default and proposed schedulers for different topologies in Scenario 1 (small cluster)

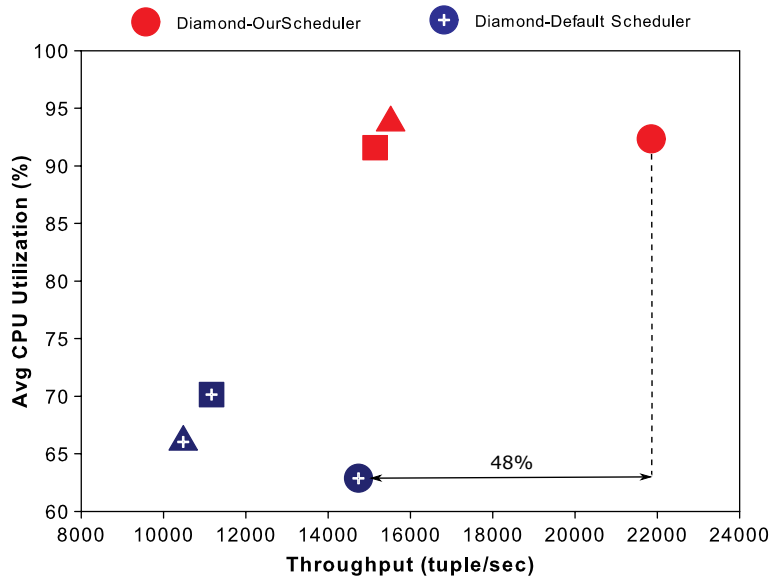

Fig. 19 Simulation results comparison of default and proposed schedulers for different topologies in Scenario 2 (medium cluster)

$$U = \sum_{i=1}^{T} x_i \bar{u_i} \tag{7}$$

In this equation, U is the overall utilization of topology on the target cluster with T types of machines. Here $\bar{u_i}$ is the average CPU utilization of all machines of type i and x_i is calculated as follow:

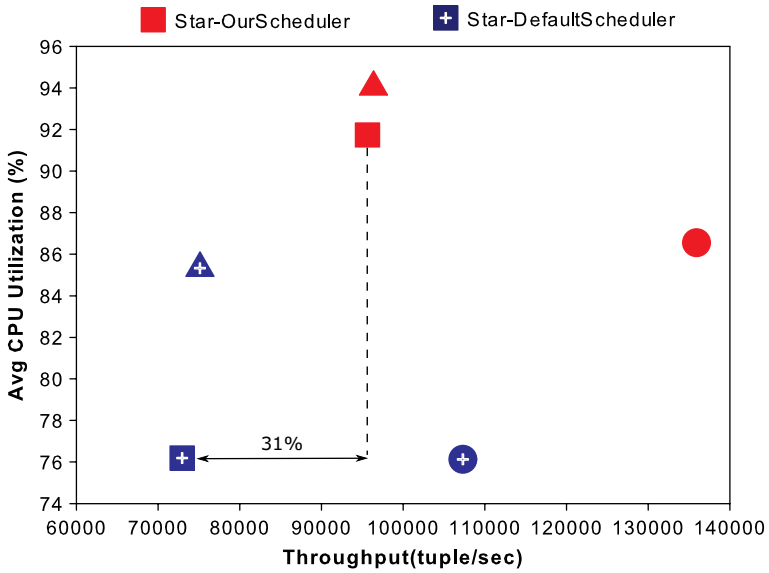

Fig. 20 Simulation results comparison of default and proposed schedulers for different topologies in Scenario 3 (large cluster)

Table 5 Ratio of differences of throughputs to utilizations between our scheduler and default scheduler.

Scenario	Linear	Diamond	Star
1	1.41	2.67	1.15
2	1.15	1.03	1.17
3	2.68	1.95	1.52

$$x_i = \sum_{j=1}^{C} x_{ij} \tag{8}$$

where

$$x_{ij} = \frac{\frac{1}{e_{ij}}}{\sum_{k=1}^{T} \left(\frac{1}{e_{ik}}\right)}$$

We use Eq. 8 for each type of machine to determine its weight. Variable C shows how many types of components the topology has; it is different from n, which shows the total number of topology components, because a topology may have several components with the same type ($C \leq n$).

Figures 18, 19 and 20 show both throughput and utilization comparison of the proposed and default scheduler for three different topologies. According to these simulating results, for the small cluster (scenario 1), our scheduler has 26% to 49% throughput gain compared to the default scheduler while it better utilizes the processing resources

10% to 35%. In the medium cluster (scenario 2), the proposed scheduler eventuates 36% to 48% improvement for overall throughput and 31 to 47% more resource efficiency. Finally, for the large cluster (scenario 3), the proposed scheduler, in comparison with the default scheduler, provides 27% to 31% and 10% to 21% gain for throughput and utilization, respectively.

Note that higher CPU utilization is not an advantage for our scheduler. However, when the ratio of differences of throughputs to utilizations between our scheduler and default scheduler is greater than 1, it can be considered an advantage. It shows how efficiently our proposed scheduler uses the processing resources. According to the mentioned above results, these ratios are calculated and listed in Table 5. For example in case of Scenario 1, for Linear topology this ratio is calculated by dividing the *diff_thpt* by the *diff_util* (shown in Fig. 18).

Conclusions and future directions

In the age of heterogeneous computing, it is essential to consider the different computing power of processors in a distributed environment. Apache Storm has a default scheduler that assigns tasks to processing elements regardless of their capabilities. Therefore it results in the under-utilization of processing resources in heterogeneous clusters.

To overcome this deficiency, we proposed a scheduling algorithm that considers the different computing power of heterogeneous machines. Our heterogeneity-aware algorithm tries to maximize overall throughput by generating a fitting execution topology graph for a given cluster. It also guarantees that no machine will be over-utilized when running the topology. During any change in the cluster state, this algorithm can be used to recalculate the new number of instances and their suitable assignment. Nevertheless, some Storm limitations prevent achieving the best throughput or appropriate resource utilization in heterogeneous systems. As our experiments show, one of the most significant obstacles to gaining maximum CPU utilization is the simple grouping strategies of Storm. Beside of that, our model also relies heavily on profiling data. This data can be captured by running the task on each machine or by estimation based on previous runs on the cluster. More accurate profiling data, provides better prediction and hence more accurate scheduling decisions.

Our future work mainly aims at possible improvements of the scheduler efficiency, addressing the obtaining profiling data problem, and addressing the issues related to the load balancing. To handle the load balancing problem, we are implementing an intelligent grouping method that determines adequate rates for each task, depending on the computation power of the machine on which the task is running.

Author contributions

HN and MG devised the project, the main conceptual ideas, and the proof outline. SN and AD performed the implementations for the suggested experiment. HN and MG verified the results and wrote the manuscript. All authors read and approved the final manuscript.

Declarations

Competing interests
The authors declare that they have no competing interests.

Received: 21 August 2022 Accepted: 17 May 2023
Published online: 17 June 2023

References

1. "Apache Storm, distributed and fault-tolerant real-time computing." https://storm.apache.org/. Accessed 25 Nov 2016.
2. Toshniwal A, et al. "Storm@twitter," Proc. 2014 ACM SIGMOD Int Conf Manag data - SIGMOD '14; 2014;147–156.
3. Aniello L, Baldoni R, Querzoni L. Adaptive online scheduling in storm, Proc. 7th ACM Int Conf Distrib event-based Syst - DEBS 13; 2013;207.
4. Xu J, Chen Z, Tang J, Su S. T-storm: Traffic-aware online scheduling in storm, Proc - Int Conf Distrib Comput Syst. 2014;535–544.
5. Eskandari L, Huang Z, Eyers D. P-Scheduler: Adaptive Hierarchical Scheduling in Apache Storm Leila. Proc Australas Comput Sci Week Multiconference - ACSW 16; 2016;1–10.
6. Peng B, Hosseini M, Hong Z, Farivar R, Campbell R. R-Storm: Resource-Aware Scheduling in Storm, Proc 16th Annu Middlew Conf - Middlew '15; 2015;149–161.
7. Gedik B, Schneider S, Hirzel M, Wu KL. Elastic scaling for data stream processing. IEEE Trans Parallel Distrib Syst. 2014;25(6):1447–63.
8. B. Lohrmann, P. Janacik, and O. Kao, Elastic Stream Processing with Latency Guarantees, Proc - Int Conf Distrib Comput Syst, 2015–July, 399–410 (2015)
9. Xu L, Peng B, Gupta I. Stela: Enabling stream processing systems to scale-in and scale-out on-demand, Proc. - 2016 IEEE Int. Conf. Cloud Eng. IC2E 2016 Co-located with 1st IEEE Int. Conf. Internet-of-Things Des. Implementation, IoTDI 2016; 2016;22–31.
10. Rychly Marek, Skoda Petr, Smrz Pavel. Heterogeneity-aware scheduler for stream processing frameworks. IJBDI. 2015;2:70–80.
11. Goudarzi M. Heterogeneous architectures for big data batch processing in mapreduce paradigm. IEEE Trans Big Data. 2017;7790(c):1.
12. Singh MP, Hoque MA, Tarkoma S. A survey of systems for massive stream analytics, arXiv preprint arXiv:1605.09021; 2016.
13. Nasiri H, Nasehi S, Goudarzi M. Evaluation of distributed stream processing frameworks for IoT applications in smart cities. J Big Data. 2019;6(1):52.
14. Hasan K, Grounds N, Antonio J. Predicting CPU availability of a multi-core processor executing concurrent java threads. Singapore: World-Comp.Org; 2009.
15. Zhang M, Zhong S, Storm Benchmark. https://github.com/intel-hadoop/storm-benchmark. Accessed 01 Apr 2016.
16. IBM. Gathering information with the collector tool. http://www.ibm.com/support/knowledgecenter/SSEQTP-8.5.5/com.ibm.websphere.base.doc/ae/ttrb-runct.html. Accessed 01 Feb 2017.
17. Chronaki Kallia, Rico, Alejandro, Badia, Rosa M, Ayguadé Eduard, Labarta Jesús, Valero Mateo. Criticality-aware dynamic task scheduling for heterogeneous architectures, Proceedings of the 29th ACM on International Conference on Supercomputing; 2015;329-338.
18. Sun Dawei, Yan Hongbin, Gao Shang, Liu Xunyun, Buyya Rajkumar. Rethinking elastic online scheduling of big data streaming applications over high-velocity continuous data streams. J Supercomput. 2018;74:615–36.
19. Liu Xunyun, Buyya Rajkumar. Resource management and scheduling in distributed stream processing systems: a taxonomy, review and future directions. ACM Comput Surv. 2018;1:1.
20. Yue Shasha, Ma Yan, Chen Lajiao, Wang Yuzhu, Song Weijing. Dynamic DAG scheduling for many-task computing of distributed eco-hydrological model. J Supercomput. 2019;75:510–32.
21. Choi Hong Jun, Son Dong Oh, Kang Seung Gu, Kim Jong Myon, Lee Hsien-Hsin, Kim Cheol Hong. An efficient scheduling scheme using estimated execution time for heterogeneous computing systems. J Supercomput. 2013;65:886–902.
22. Hidalgo N, Wladdimiro D, Rosas E. Self-adaptive processing graph with operator fission for elastic stream processing. J Syst Softw. 2016;0:1–12.
23. Li T, Tang J, Xu J. Performance modeling and predictive scheduling for distributed stream data processing. IEEE Trans Big Data. 2016;2(4):353–64.
24. Wijbrandi W, Meijer RJ, Van Der Veen JS, Van Der Waaij B, Lazovik E. Dynamically Scaling Apache Storm for the Analysis of Streaming Data, IEEE Conference on Big Data Computing Service and Applications (BigDataService); 2015. p. 154–161.
25. Liu Xunyun, Buyya Rajkumar. D-Storm: Dynamic Resource-Efficient Scheduling of Stream Processing Applications. IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS). IEEE; 2017.
26. Eskandari Leila, et al. Iterative Scheduling for Distributed Stream Processing Systems. Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems. ACM; 2018.
27. Inggs Gordon, Thomas David B, Luk Wayne. A domain specific approach to high performance heterogeneous computing. IEEE Trans Parallel Distrib Syst. 2017;28(1):2–15.
28. Chekuri Chandra, Khanna Sanjeev. A polynomial time approximation scheme for the multiple knapsack problem. SIAM J Comput. 2005;35(3):713–28.

29. Nasiri H, Nasehi S, Goudarzi M. A survey of distributed stream processing systems for smart city data analytics, In: Proceedings of the international conference on smart cities and internet of things, ACM; 2018;12.
30. Nasiri H, Goudarzi M. Dynamic fpga-accelerator sharing among concurrently running virtual machines. In: 2016 IEEE East-West Design & Test Symposium (EWDTS), IEEE; 2016;1-4.
31. Storm Heterogeneity-aware Scheduler. https://github.com/h-nasiri/Storm-Heterogeneity-aware-Scheduler; 2021.
32. Scheduling Simulator. https://github.com/h-nasiri/Scheduling-Simulator; 2022.
33. Kavand N, Darjani A, Nasiri H, Goudarzi M. Accelerating distributed stream processing, United States Patent 10534737; (Feb. 14, 2020).
34. Farahnakian F, Liljeberg P, Plosila J. LiRCUP: Linear Regression Based CPU Usage Prediction Algorithm for Live Migration of Virtual Machines in Data Centers, 2013 39th Euromicro Conference on Software Engineering and Advanced Applications, Santander, Spain; 2013;357-364, https://doi.org/10.1109/SEAA.2013.23.
35. Kudinova Marina, Melekhova Anna, Verinov Alexander. CPU utilization prediction methods overview. In: Proceedings of the 11th Central & Eastern European Software Engineering Conference in Russia (CEE-SECR '15). Association for Computing Machinery, New York, NY, USA, Article 7; 2015;1–10.
36. Gupta S, Dileep AD, Gonsalves TA. A joint feature selection framework for multivariate resource usage prediction in cloud servers using stability and prediction performance. J Supercomput. 2018. https://doi.org/10.1007/s11227-018-2510-7.
37. Huang Z, Peng J, Lian H, Guo J, Qiu W. Deep recurrent model for server load and performance prediction in data center. Complexity. 2017. https://doi.org/10.1155/2017/8584252.
38. Chen S, Shen Y, Zhu Y. Modelling conceptual characteristics of virtual machines for CPU utilization prediction. Concept Modell. 2018. https://doi.org/10.48550/arXiv.1811.04731.

Publisher's Note