The final publication is available at https://link.springer.com/article/10.1007/s11227-020-03223-z
The Journal of Supercomputing

Job Scheduler for Streaming Applications in Heterogeneous Distributed Processing Systems

Ali Al-Sinayyid and Michelle Zhu

Abstract In this study, we investigated the problem of scheduling streaming applications on a heterogeneous cluster environment and, based on our previous work, developed the maximum throughput scheduler algorithm (MT-Scheduler) for streaming applications. The proposed algorithm uses a dynamic programming technique to efficiently map the application topology onto the heterogeneous distributed system based on computing and data transfer requirements, while also taking into account the capacity of the underlying cluster resources. The proposed approach maximizes the system throughput by identifying and minimizing the time incurred at the computing/transfer bottleneck. The MT-Scheduler supports scheduling applications structured as a directed acyclic graph (DAG). We conducted experiments using three Storm microbenchmark topologies in both simulation and real Apache Storm environments. In terms of the performance evaluation, we compared the proposed MT-Scheduler with the simulated round robin and the default Storm scheduler algorithms. The results indicated that the MT-Scheduler outperforms the default round robin approach in terms of both the average system latency and throughput.

Keywords Apache Storm, DataStream, Distributed systems, Heterogeneous scheduling, DAG scheduling

1 Introduction

At present, we live in the big data era, in which a variety of applications such as stock trading, banking systems, healthcare databases, IoT sensors, and social media networks [1] generate colossal amounts of real time data. Such distributed data stream processing systems (DDSPSs) usually compute unbounded streams of data in real time, and they are dynamic in terms of their resource capacities [4-5]. To realize such continuous data generation via streaming applications, the underlying distributed processing systems must perform prompt yet efficient management and analysis, especially in the case of heterogeneous systems [2-3]. One of the key objectives of scheduling streaming applications is to maximize the frame rate, which corresponds to the number of instances of the datasets that can be processed per unit time. To achieve this goal, the scheduling algorithm must consider the data locality, resource heterogeneity, communicational aspects, and computational latencies.

Data locality and location awareness factors arise due to the high data transfer latency in cases in which the data sources often reside in distant DDSPSs, which can negatively impact the system performance [6]. Researchers have addressed and solved this problem by performing the computing as close as possible to the data source [7]. An efficient mapping strategy should thus constrain the significant data traffic onto the same machine or nearby machines to minimize the communication time and mitigate the data transferring latencies

Furthermore, the presence of cluster heterogeneity in a distributed environment results in different capabilities for task execution and data transmission, because of which, the related scheduling algorithm pertains to an NP-complete problem [43-45]. Both heterogeneous DDSPSs and job applications can have a variety of resource capacities and task complexities, respectively [8]. Consequently, a scheduling approach that does not consider the aspects of resource heterogeneity and task complexity variation in communication and computation might impact the performance and reduce the system frame rate [9-12].

In this work, we aim to minimize the bottleneck time for the transfer time and node computing time along the execution path to achieve the maximal frame rate for the streaming applications. The main contributions can be summarized as follows:

I. We propose a maximum throughput scheduling algorithm named MT-Scheduler to maximize the system throughput by using dynamic programming. The maximization is performed by strategically assigning the task components to the appropriate nodes based on their computational and communicational requirements, based on our previous work [44]. The MT-Scheduler supports scheduling applications that are structured as a DAG, such as Amazon Timestream, Google Millwheel, Yahoo S4, and Twitter Heron [15-18], [47-49].

II. We implement the MT-Scheduler algorithm in a simulation environment. The testing results show that the MT-Scheduler can significantly improve the system throughput compared with the corresponding performance of the simulated round robin algorithm.

III. Furthermore, we implement the MT-Scheduler in Apache Storm 0.9.7 [19] with a cluster of 8 heterogeneous physical machines. For the evaluation, we test three well known microbenchmark topologies [26-28,38], specifically, the linear, star, and diamond topologies. The results are compared with those for the default Apache Storm scheduler and an adaptive online scheduler [20]. The test results indicate that the MT-Scheduler outperforms both the schedulers in terms of the system latency and frame rate.

IV. We propose a polynomial time heuristic solution to a known NP-complete problem [43-45] by utilizing the dynamic programming technique in our MT-Scheduler algorithm.

V. The proposed scheduling algorithm covers the knowledge gap in the existing literature, corresponding to both the cluster and topology characteristics as scheduling parameters, in addition to transparently allowing the user to control the data locality aspect.

The remaining paper is organized as follows. Section 2 provides a review of the related works. Section 3 presents the mathematical model for the system and the scheduling problem formulation. Section 4 describes the MT-Scheduler algorithm. Sections 5 and 6 present the evaluation results obtained using the simulation and real environment experiments, respectively. Finally, Section 7 concludes the paper and discusses future work.

2 Related Work

Extensive research on scheduling strategies for distributed streaming processing systems has been performed [6, 13, 14, 21-24]. Most of the proposed algorithms aimed to improve the system performance by reducing the time and cost incurred by scheduling. In Apache Storm [19], a simple round robin (RR) was used as the default scheduler [25]; however, a satisfactory performance was not ensured. In addition, several Storm scheduler algorithms have been proposed to optimize the system performance.

Aniello et al. [20] proposed two types of scheduling algorithms for Storm, namely, offline and online schedulers, using which, the tuple transfer latency between the components could be reduced. The offline scheduler identified the most connected components from the job DAG topology and mapped them to the same node. During runtime, the online scheduler monitored the tuple transfer latency and adjusted the mapping schema accordingly by using a best fit greedy approach to minimize the interslot and internode traffic. In this approach, each component task pair was examined separately from the other topology components, likely resulting in two extensively communicating components being mapped to different nodes.

Peng et al. [26] proposed an offline resource aware scheduler, namely, R-Storm, to achieve the maximum throughput and resource utilization within the user predetermined resource budget. This algorithm conducts topological sorting by using the breadth first search (BFS) principle to minimize the internode traffic latency. Later, the input information specified by the users regarding the resource constraints are passed as parameters to a quadratic multiple 3D knapsack problem. The R-Storm can outperform the default scheduler; however, the users are extensively involved in this process.

Likewise, a traffic aware scheduler named T-Storm [27] was used to minimize the internode and interprocess traffic**.** This solution, in contrast to R-Storm, was transparent to users; however, the intercommunication between the tasks was ignored.

Cardellini et al. [29-30] and Nardelli et al. [31-32] performed task scheduling over geographically distributed heterogeneous clusters under the QoS constraints. The network aware scheduling algorithm proposed by these researchers minimized the network traffic and improved the system efficiency in terms of the communication latency, cluster resource utilization, and application availability.

Li et al. [33] proposed a scheduling strategy by implementing the dynamic topology adjustment for Apache Storm. The topology optimization enabled the identification of the performance bottlenecks by examining the bolt capacity and the incoming/outgoing tuple transfer queue.

Zhang et al. [34] developed a latency aware edge computing platform built on Apache Storm. This approach could be used to minimize the end to end latency in the case of a heterogeneous network and node resources (GPUs and CPUs).

Liu et al. [35] presented a heuristic scheduling algorithm for Apache Storm, in which the historical traffic latencies and task topology were used to predict the system performance. The tuple processing latency and tuple failure rate were reduced by identifying the overloaded node for task migration. However, this algorithm could only function in a homogenous cluster.

Shukla and Simmhan [36] proposed a heuristic algorithm that used a model driven approach from the queueing theory for the resource allocation prediction and task mapping to maximize the throughput. The same task threads were allocated and scheduled in the same machine or adjacent nodes to reduce the intercommunication and achieve the peak data rate.

Kombi et al. introduced [37] a holistic approach (DABS-Storm) that adapted the task requirements by dynamically controlling the resource usage as a latency aware load balancing strategy in stream processing systems.

Eskandari et al. [38] presented an online scheduler based on the topology DAG partition as an extension to their P-Scheduler [28]. The algorithm aimed to minimize the data transfer and maximize the resource utilization by considering the network and task characteristics. In addition, Liu et al. [39] proposed a dynamic resource aware scheduler named D-Storm by using a greedy algorithm to solve the bin packing problem.

Among the aforementioned scheduling strategies, most of the algorithms consider the topology structure, intercommunication traffic, or computing node load aspects. However, the heterogeneity in the task, network, and computer resources is not always considered. The proposed scheduling algorithm overcomes these limitations pertaining to the algorithms reported in the existing literature. Unlike the existing approaches, MT-Scheduler maximizes the throughput of a heterogeneous DDSPS by considering both the cluster and application characteristics as scheduling parameters. In addition, the algorithm identifies and minimizes the potential computational or communicational bottlenecks by utilizing the dynamic programming technique. Furthermore, the proposed algorithm allows the users to transparently select the sites and configure the data locality configuration.

3 Problem formulation

3.1 Problem Definition

As in our previous work [44], an underlying node cluster is modeled as a graph $Cluster(V_{\mathcal{C}}, E_{\mathcal{C}})$, with $|V_{\mathcal{C}}| = z$, where $V_{\mathcal{C}}$ denotes a cluster set that consists of z geographically distributed heterogeneous nodes (vertices) denoted as n_i where $i = 1,2,..,z$. Node n_i has an attribute of a processing power p_i. $|E_{\mathcal{C}}|$ denotes the set of cluster network links (edges), where n_i is connected to its neighbor node $n_{i_{succ}}$ with a network link of bandwidth $\ell_{i,i_{succ}}$. The transport network may or may not be a complete graph, depending on whether the node deployment environment is the Internet or a network in single or multiple distributed sites.

An application in distributed data stream processing systems such as Apache Storm [19], Apache Flink [41], Apache Spark [42], S4 Platform [17], and Twitter Heron [18] can be represented as a (DAG). Let the topology be represented as $Topology = (V_{\mathcal{TP}}, E_{\mathcal{TP}})$, where $|V_{\mathcal{TP}}| = k$ is a set of k components (vertices) $c_1, c_2, \ldots, c_k$. Component c_1 is the data source, namely, *Spout,* which reads data from an external source and transmits it as a data tuple to the successor application components. c_j, termed as $Bolt$, where $j = 2, 3, \ldots, k$, performs a computational task of complexity x_j on the incoming data sized m_{j-1}, sent from its preceding task c_{j-1}. The computational components process the data tuples received from either a source or another computational component before transmitting the processed stream to another component. $|E_{\mathcal{T}}|$ denotes a set of links (edges) that represents the dependency of the topological components and data transfer.

Based on the user preferences, all the cluster nodes n_i and topological components c_j are divided into geographical site tags $\mathcal{S}_{tag}$, $tag \in [1, tags_{total}]$, where $tag = 1,2,..tags_{total}$. After configuring the metadata, each cluster node n_i and component c_j are tagged with a metadata $\mathcal{S}_{tag}$. For $tags_{total}$ of unique metadata $\mathcal{S}_{tag}$ ID, $tags_{total}$ number of groups exist, specifically, $\mathcal{S}_1, \mathcal{S}_2, .., \mathcal{S}_{total}$. Each group $\mathcal{S}_{tag}$ consists of a user predetermined tasks and b nodes with the same metadata value $\mathcal{S}_{tag} = [c_{\theta_1}, c_{\theta_2}, .., c_{\theta_a}, n_{\theta_1}, n_{\theta_2}, .., n_{\theta_b}]$, where $\emptyset \in tag$.

Fig. 1 shows one of the possible undesirable scenarios that can be caused by implementing the round robin algorithm. The application has different components to be mapped to a heterogeneous cluster. Due to the even distribution strategy, the RR scheduler may assign C2, which is a CPU intensive task component to machine N1, although other machines with a higher processing power are available. Furthermore, assigning an I/O intensive task between C4 and C5 to nodes from different sites (N3 and N4) might incur a larger networking delay.

3.2 Objective Function

Based on our previous work [44], the system performance optimization and throughput rate maximization can be realized by identifying and minimizing the potential performance bottleneck, in terms of both the computational and communicational latencies.

The computational task complexity x_j is a parameter that determines the CPU bound jobs and the associated computational logic complexity as a data operator. Correspondingly, this parameter helps indicate the processing power necessary to compute a function of a task c_j for its incoming data sized m_{j-1}. The output data with a size of m_{j+1} is in turn transferred to the incoming message queue of its succeeding component c_{j+1} for further processing. The processing power of a node n_i in a heterogeneous cluster p_i represents the assigned executors' capability for processing n_i. Therefore, we can estimate the average computing time $\mathcal{T}_{compute}$ for task c_j on a node n_i as follows:

$$\mathcal{T}_{compute}(n_i, c_j) = \frac{c_j(x(m_{j-1}))}{n_i(p)} \quad (1)$$

The estimated computing time $\mathcal{T}_{compute}(c_j, n_i)$ is the average time required to compute a task c_j with a computational complexity of x_j for tuple data sized m_{j-1}, which is executed on a supervisor node n_i with an executor of processing power p_i, to produce a fully processed data unit. Practically, in the Storm environment, this time refers to the time period that starts as soon as the Storm *_execute*() method is called, which executes the required job of the task, and ends when the tuple is fully processed and ready to be transferred to the next subscribed components. In a heterogeneous cluster, the execution latency varies from high, as a potential bottleneck, to low. This latency depends on the task complexity and its tuple size, as well as the assigned executor processing power.

In DDSPSs, the tasks are communicated through the transfer of messages over the underlying network links. $\ell_{i,i_{succ}}$ denotes the bandwidth of the transferring link that transfers a data tuple of size m_j between node n_i and its successor node $n_{i_{succ}}$.

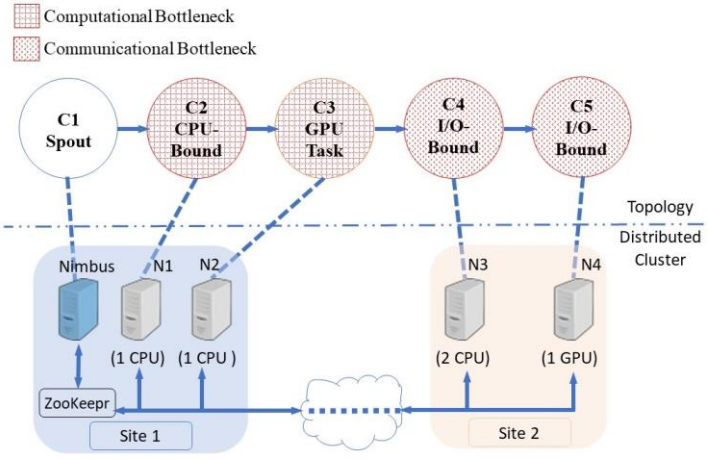

Figure 1 Default Storm scheduler that does not take into account the data locality or performance bottleneck.

We can compute the estimated average transfer time $\mathcal{T}_{transfeer}$ as

$$\mathcal{T}_{transfeer}(\ell_{i,i_{succ}}, m_j) = \frac{m_j}{\ell_{i,i_{succ}}} \quad (2)$$

The estimated tuple transfer latency $\mathcal{T}_{transfeer}$ is the average time to transfer an already processed tuple from the outgoing buffer of one component to its successor incoming queue.

The proposed mapping scheme divides the cluster nodes n_i and topological components c_j into user defined geographical site tags $\mathcal{S}_{tag}$. Next, for each $\mathcal{S}_{tag}$, a group of a components and b nodes are used to combine the topological components into q groups of tasks denoted by $g_1, g_1, \ldots, g_q$. These tasks are mapped onto a selected network path P of q supervisors

within the $\mathcal{S}_{tag}$ from n_s to n_d in the Storm cluster network, where $n_s, n_d \in |V_C|$ and $q \in \left(min(k, z), min(a, b)\right)$. The potential scheduling path P consists of a series of nodes, which are not necessarily distinct supervisors, based on the metadata configuration. The bottleneck time for each site tag $\mathcal{T}_{bottleneck[\mathcal{S}_{tag}]}$ is the maximum required time by the distributed data stream processing system to compute and transfer a data unit (fully processed) by a time unit. The objective function of identifying and minimizing the bottleneck $\mathcal{T}_{bottleneck[\mathcal{S}_{tag}]}$ can be defined as in equation (3):

$$\mathcal{T}_{bottleneck[\mathcal{S}_{tag}]}(\text{Path } P \text{ of } q \text{ nodes}) = \max_{\substack{Path\ P\ of\ q\ nodes \\ i=1,2,\dots,q}} \left\{ \begin{matrix} \mathcal{T}_{compute}(\mathcal{g}_i), \\ \mathcal{T}_{transfeer}\left(\ell_{P_{[i],P_{[isucc]}}}\right) \end{matrix} \right\}$$

$$= \max_{\substack{Path\ P\ of\ q\ nodes \\ i=1,2,\dots,q}} \left\{ \begin{matrix} \frac{1}{\mathcal{p}_{P[q]}} \sum_{j \in \mathcal{g}_i, j \geq 2} \left(x_j\left(m_{j-1}\right)\right), \\ \frac{m(\mathcal{g}_i)}{\ell_{P_{[i],P_{[isucc]}}}} \end{matrix} \right\} \quad (3)$$

4 Proposed MT-Scheduler

Achieving an optimal solution to the considered scheduling problem by maximizing the tuple processing rate can be difficult and computationally infeasible; thus, a simple yet effective algorithm is required. The scheduling of DAG jobs in a distributed stream processing system with different job requirements corresponds to an NP-complete problem [43-45]. Thus, we propose a high throughput scheduler for distributed data stream processing systems, based on our previous work [44]. The MT-Scheduler algorithm, considers, in addition to metadata groups, the topology job and node attributes, including the computational complexity, data size, node processing power, and link transfer bandwidth. The proposed algorithm achieves the maximum tuple processing rate by utilizing a dynamic programming technique for job mapping, which recursively minimizes the time incurred on the bottleneck and provides a polynomial time solution. The maximal frame rate that a system can achieve is limited by the slowest element (bottleneck) in the transport link or computing node along the cluster. This work proposes two algorithms, namely, Algorithm 1 (mapper), which is the main algorithm, and Algorithm 2, which is the MT-Scheduler (for linear critical path mapping).

The mapper algorithm, expressed as Algorithm 1, inputs the data details for the submitted topology (ID, Name, Submitted user) and the underlying cluster (nodes, worker slots, and executors). First, the directed acyclic graph topology is linearized by implementing a topological sorting process. Next, the critical path is identified by using the well-known polynomial Longest path algorithm (LP). The linear critical path represents the most time consuming sequence of topological components that the system must implement sequentially. Please note that we assume a homogenous network when identifying the critical path using this method. Although this case is not realistic, we adopt this assumption for simplification. Next, the mapper algorithm calls Algorithm 2 to determine the mapping schema for the topological components in the critical path $\mathcal{CP}$. The topological components not on the critical path $\mathcal{CP}$ are mapped using a simple layer oriented greedy method. We apply a topological sort to order the non-$\mathcal{CP}$ components into layers and sort these components in a descending order based on the $\mathcal{T}_{compute}$ and $\mathcal{T}_{transfeer}$. The components that require more computations and communications are assigned higher priorities. Subsequently, we map the components linearly layer by layer; the component with a higher priority is mapped to a node with higher resources. In the Storm cluster, two types of nodes exist: The master, which runs a daemon named Nimbus, and worker nodes that run a daemon named Supervisor. Nimbus periodically calls the scheduler to update the mapping process. The mapper algorithm verifies the topology scheduling if required, to avoid repetitive scheduling implementation and system overloading. Finally, the mapper algorithm utilizes the pluggable scheduler feature in Storm, and,

Algorithm 1 Mapper *implements IScheduler* interface in Storm Nimbus.

Input: $\mathcal{TP}$ as the submitted task topology
$Cluster$ as the underlying $Cluster$_Details (supervisors n, workerslots w, executors ex),
Output: Implement final scheduling schema

$\mathcal{TP} \leftarrow$ TopologyDetails.get();
$Cluster \leftarrow$ SupervisorDetails.get();
Critical Path $\mathcal{CP} \leftarrow$ Extract Critical Path ($\mathcal{TP}$) ;
CP_HashMapping ←MT-Scheduler (Critical Path $\mathcal{CP}, Cluster$);
NCP_HashMapping ←Map noncritical path tasks using layer based greedy algorithm;
Final_ HashMapping ← Join (CP_HashMapping, NCP_HashMapping)
if ($\mathcal{TP}$ needs_Scheduling ==True) **then**
get $\mathcal{TP}$'s tasks as (c);
for each c_i in Final_ HashMapping **do** // Assign all tasks to supervisor workers and executers in the mapped node
find corresponding n in Final_ HashMapping
if ((supervisor workers $n.w \neq Null$) **AND**
(supervisor executers $n.ex \neq Null$) **then**
Cluster.Assign (c_i , n);
end if
end for
end if

via the Nimbus node, implements the final mapping schema by assigning all the critical and noncritical path components for the submitted topology $\mathcal{TP}$ to the underlying $Cluster$.

In Algorithm 2, the input for the MT-Scheduler is the underlying cluster data details along with the critical path list and $MetaKeys\{Stag\}$ as the user defined data list of the site ID/tag. First, the algorithm generates Tags_Pairs, which is a list of critical path pairs, with each pair consisting of a node and a task belonging to the critical path set $\{(node, task) \in \mathcal{CP}\}$. Through the dynamic programming technique, the MT-Scheduler recursively chooses a critical topology path based on the previous round of calculation. At each step of the recursion, the algorithm maps the partial components pipeline to the underlying network nodes and calculates the new potential mapping cost.

The recursion process in the MT-Scheduler algorithm continues until the mapping results converge to a mapping scheme that achieves the objective and minimizes the system bottleneck for the critical path components in the submitted application.

Equation (4) presents the recursion based on dynamic programming, which leads to a potential mapping for the critical path components in the MT-Scheduler algorithm. Let $1/\mathcal{P}map^{j}(n_i)$ denote the maximal tuple rate with the first j topology components mapped to a path from a source node n_s to a node n_i in an arbitrary computer network. Let $S^{J}(n_i)$ represent the sum of the tuple sizes of all the components on a node n_i with the first *j* tasks mapped from node n_s to n_i in metadata group $\mathcal{S}_{tag}$. Consequently,

$$\underset{j=1\ to\ k,\ \ n_i \in V,\ \ tag=1\ to\ total}{\mathcal{P}map^{j}(n_i)_{[\mathcal{S}_{tag}]}} = min\left\{ \begin{array}{l} max\left(\begin{array}{c} \mathcal{P}map^{\,j-1}(n_i), \\ \mathcal{T}_{comput}\left(x_{j+1}\left(S^{\,j-1}(n_i) + (m_j)\right), \mathcal{P}_{n_i}\right) \end{array} \right) \\ \underset{u \in adj(n_i)}{min} \left(max\left(\begin{array}{c} \mathcal{P}map^{\,j-1}(u), \\ \mathcal{T}_{comput}\left(x_{j+1}(m_j), \mathcal{P}_{n_i}\right), \\ \mathcal{T}_{transfeer}\left(m_j, \ell_{u,n_i} \right) \end{array} \right) \right) \end{array} \right\}$$

$$= min\left\{\begin{array}{l} max\left(\begin{array}{c}\mathcal{P}map^{\,j-1}(n_i),\\ \frac{\left(S^{\,j-1}(n_i)+\,x_{j+1}(m_j)\right)}{\mathcal{P}_{n_i}}\end{array}\right) \\ \min\limits_{u\in adj(n_i)}\left(max\left(\begin{array}{c}\mathcal{P}map^{\,j-1}(u),\\ \frac{(\,x_{j+1}(m_j))}{\mathcal{P}_{n_i}},\\ \frac{m_j}{\ell_{u,n_i}}\end{array}\right)\right)\end{array}\right\} \quad (4)$$

with the base conditions computed as

$$\underset{n_i\in V\ and\ n_i\neq n_s}{\mathcal{P}map^1(n_i)} = \begin{cases} max\left(\frac{x_2\,(m_1)}{\mathcal{P}_{n_i}}, \frac{m_1}{\ell_{n_s,n_i}}\right) & \forall e_{n_s,n_i}\in E \\ +\infty & otherwise. \end{cases}$$

and $\mathcal{P}map^t(n_s) = \sum_{i=1}^{t}(\,x_{i+1}\,m_i/\mathcal{P}_s)$ *where* t=1, 2,..,k

Algorithm 2 MT-Scheduler (Critical Path)

Input: Critical path, Cluster (V_C, E_C), $MetaKeys\{Stag\}$ as the user defined list of site ID/tag
Output: MTPR_HashMapping< $component\ c, node\ n$ > //Scheduling mapping schema between node and task component

Generate *Tags_Pairs*$_{stag}$ (c, n); //by pairing c, n with the same site tag (*Stag*)
for each $v_i \in V_C$ with only c_1 **do** //Initialize the 2D matrix and calculate the base condition
  if $e_{1,i} \in E_C$ **then**
    $MappingMetrix_{[i,1]}$ ← Calculate $\mathcal{P}map^1(n_i)$;
  else $MappingMetrix_{[i,1]} = +\infty$;
  end if
end for
for each $Stag \in MetaKeys$ **do**
  for each $c \in$ *Tags_Pairs*$_{stag}$ **do**
    for each $n \in Tags\_Pairs_{\text{stag}}$ **do**
      if c_{j-1} mapped to n_i **then** //case I
        Map c_j to supervisor n_i ;
        $BT1$ ← Calculate $\mathcal{P}map^j(n_i)$; //case II
        for each $adj(n_i)$ directly connected to n_i **do**
          Map task c_j to n_i;
          $BT_{ad\ j(n_i)}$ ← Calculate $\mathcal{P}map^j(adj(n_i))$;
        end for
      end if
    $BT2 \leftarrow min\left(BT_{ad\ j(n_i)}\right)$;
    $MappingMetrix_{[i,j]} \leftarrow min(BT1, BT2)$; //choose minimum BT among 2 cases
    end for
  end for
  MTPR_HashMapping ← MTPR_HashMapping $(c_{j_{tag}}, n_{i_{tag}}).add$;
end for
return MTPR_HashMapping.

Every link, node, or task is a potential bottleneck and needs to be checked. The recursive dynamic programming process expressed in Equation 4 generates a 2D matrix [44]. As shown in Algorithm 2, after calculating the recursion base conditions, at each step of the recursion process, the bottleneck times are calculated for all potential mapping schemas, and the minimum time is selected to achieve the maximum frame rate.

In a deployment over multiple sites, it may be essential to allow users to assign a particular topology component to a specific supervisor located at a specific site. However, Storm users, by using the default scheduler, cannot predict the mapping of the topological components in the Storm cluster. The MT-Scheduler allows the users to configure and regulate

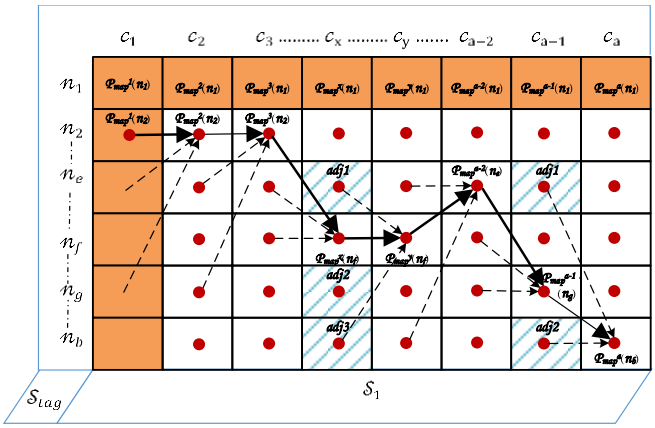

Figure 2 MT dynamic programming and 2D matrix construction [44]

the data locality aspects by utilizing the metadata configurations of the Storm nodes to execute tasks as close to the data as possible, which leads to the minimization of the transfer cost. In Apache Storm, the users can transparently establish the metadata configuration by setting the *supervisor.scheduler.meta* Storm field in each supervisor's configuration file to specify the custom site tags. After tagging the supervisors, the users can tag the components accordingly to ensure that the scheduler can correctly associate the spouts/bolts with the supervisors. The Storm method *addConfiguration* processes the tagging configuration to allow the user to build the topology stage. The metadata for each supervisor can be obtained by calling the Storm method *getSchedulerMeta*, which returns the metadata in key-value pairs. By default, if no site configuration is specified the user, MT-Scheduler considers all the tasks and nodes tagged as one single group.

In Fig. 2, each cell $\mathcal{P}map^{j}(n_i)$ in the matrix represents a partial mapping solution that maps the first j tasks to a path between n_s and n_i, where both nodes have the same $\mathcal{S}_{tag}$. Each iteration step involves the calculation of the bottleneck value to fill in a new cell $\mathcal{P}map^{j-1}(n_i)$ and add new tasks to the partial scheduling schema.

In the 2D matrix process, we consider two subcases, the minimum value of which is chosen as the minimum $\mathcal{T}_{bottleneck}[\mathcal{S}_{tag}]$. These cases can be described as follows. Case I: The new task is mapped to the same node that has executed the previous task. We directly place component c_j at supervisor n_i, at which the last task c_{j-1} was executed in the previous mapping subproblem $\mathcal{P}map^{j-1}(n_i)$. In other words, the last two or more components are scheduled to the same node n_i to minimize the internode communication latency. Therefore, we

only need to add the computing time $\mathcal{T}_{compute}$ of c_j on node n_i to the $\mathcal{P}map^{j-1}(n_i)$ time.

Case II: The new task is mapped to one of the neighbor nodes n_u, where $n_u \in adj(n_i)$ and has a direct link to n_i, which is represented by a dotted line from a neighbor shaded cell on the left column to the supervisor n_i. We recursively calculate $\mathcal{T}_{bottelneck}$ for all possible mappings to n_u nodes and choose the minimal value. This minimal value is further compared with the value calculated in Case 1. The minimum of these two values is selected as the minimum $\mathcal{T}_{bottelneck}$ for the partial mapping to a path between n_s and n_i with the same $\mathcal{S}_{tag}$.

For further clarification, we explain both the cases in the presented scheduling scenario in the matrix shown in Fig. 2. For scheduling the component c_y, the MT-Scheduler algorithm first calculates the bottleneck time if the component is assigned to the same node to which the previous component was mapped, and in this scenario, if the node is n_f, the case corresponds to Case I. Second, each bottleneck time is calculated if the component is assigned to one of the adjacent/neighbor nodes (shaded cells); in this scenario, the nodes are *adj1, adj2* and *adj3* (nodes n_e, n_g *and* n_b respectively), which correspond to Case II. Finally, the MT algorithm chooses the minimum bottleneck time and assigns the task to the correspondent node, namely, n_f. Another example, as shown in Fig. 2, corresponds to the scheduling of the last component c_a.

In contrast to in the previous example, instead of assigning this task to the same node executing the previous task (n_g), the algorithm chooses to assign this task to one of the n_g adjacent nodes *adj1* and *adj2* (nodes n_e *and* n_b, respectively) as in Case II. The MT algorithm calculates the minimum bottleneck time achieved when assigning task c_a to the adjacent node (n_b).

Fig. 3 shows the architecture and dataflow of the proposed scheduler. The MT-Scheduler algorithm inputs the user defined list of site tags to generate a Tags_Pairs list from the Storm metadata configurations. Next, according to the input critical path topological components and cluster characteristic data, the MT-Scheduler algorithm uses the dynamic programming

to generate MTPRHashMapping $< node, component >$ for the critical path topological components. The main mapper algorithm builds the final mapping schema by calling the MT-Scheduler for the critical path components and integrates the components with the mapping schema for the noncritical path components. Finally, the pluggable scheduler feature in Storm is utilized to implement the final mapping schema via Nimbus over the underlying cluster.

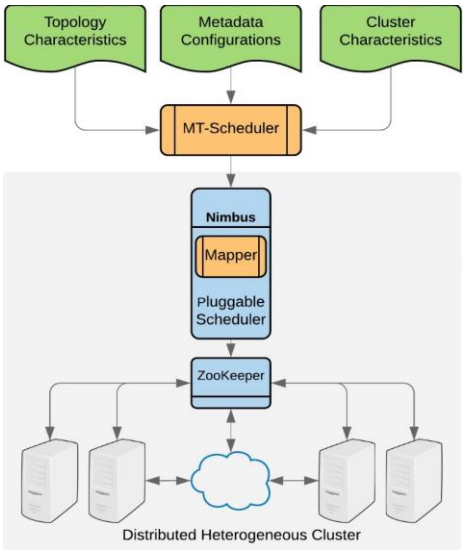

Figure 3 Architecture and dataflow of the MT-Scheduler

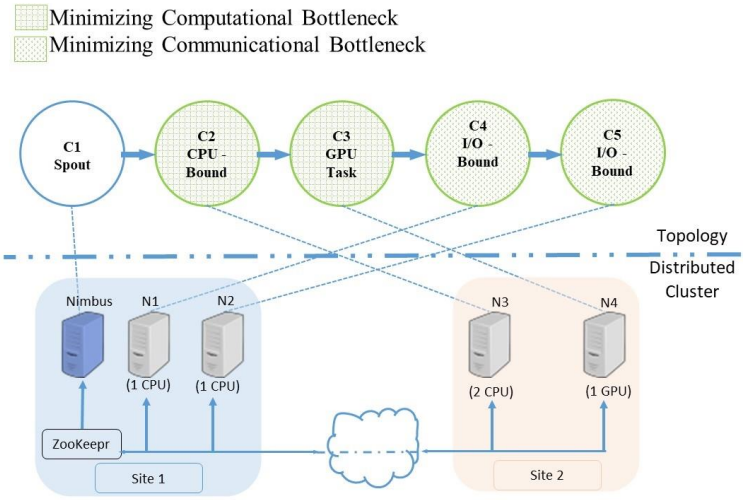

Figure 4 MT-Scheduler minimizes the system performance bottlenecks

The MT-Scheduler, as shown in Fig. 4, solves the performance bottleneck problem arising in the default scheduler shown in Fig. 1. The proposed algorithm can minimize the computational bottlenecks by assigning C2 to node N3 with sufficient processing power and allowing the user to assign a GPU node N4 in Site 2 to execute the GPU required tasks of C3. Furthermore, the algorithm minimizes the communicational bottlenecks by assigning both C4 and C5 to nodes located at the same site to minimize the internode transfer latency.

5 Simulation Results

The proposed MT-Scheduler is implemented in a simulation program, as described in our previous work [46] by using C++, and it runs on a Windows 10 machine featuring Intel(R) Core (TM) i7-8565U CPU @ 1.80 GHz, RAM 16 GB and SATA disk of 1 TB. For comparison, we implement the RR default algorithm as the Storm default scheduler in C++. Three microbenchmark topologies, namely, linear, diamond, and star topologies, are randomly generated with various computing complexities and data transfer sizes.

We conduct a simple experiment to illustrate the influence of the task parameters on the scheduling decision and performance. Scenario 1, which involves a task with low computing and networking load, and scenario 2, which involves a task with high computing and transfer load, are tested on a cluster of 8 nodes. As shown in Fig. 5, scenario 1, which has lower loads, generally achieves a higher system performance compared to that in scenario 2, which has higher loads. The highest frame rates in scenario 1, as obtained using the default RR and MT-Scheduler, are 35 and 45 frames per second, respectively. In contrast, in scenario 2, the highest frame rates, as obtained using the default RR and MT-Scheduler, are 39 and 60 frames per second, respectively. The proposed MT-Scheduler algorithm scales better than the RR. Furthermore, we test the system throughput performance when the underlying cluster size scales up. The same three task topologies are

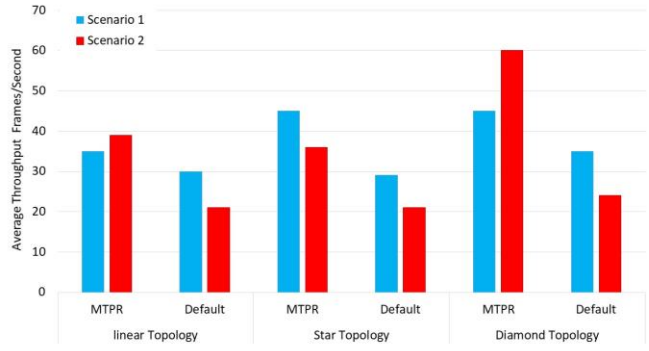

Figure 5 Impact of the computational complexity and data transfer rate in a distributed heterogeneous cluster scheduling

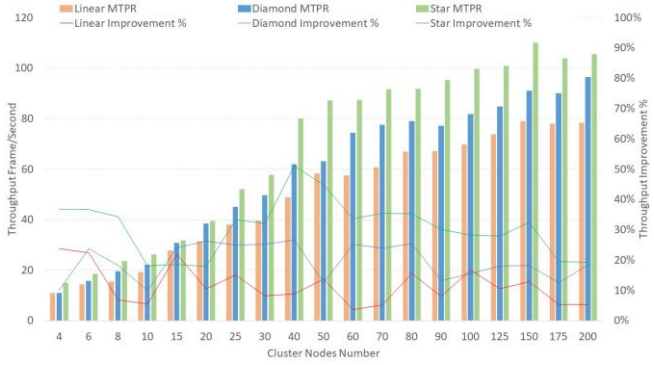

Figure 6 Simulated system average throughput, scalability and throughput improvement percentage

used, as shown in Fig. 6 in three different colors, and the number of cluster nodes ranges from 4 to 200, as shown in the x axis. Fig. 6 demonstrates that the MT-Scheduler, as indicated by the height bars, maintains higher frame rates than those obtained using the default RR, which are represented as connected curves as the cluster scales up. Out of the three linear, diamond and star topologies, the star topology scales the best.

6 Real Storm Environmental Results

Table 1 Experimental Cluster Specification

Cluster Role	(Intel(R)Core(TM))	CPU-Memory-Storage
Nimbus and ZooKeeper	i7-2600 3.40 GHz	16 GB - 2 TB
Supervisor 1	i7-2600 3.40 GHz	16 GB - 500 GB
Supervisor 2	i7-8565U 1.80 GHz	16 GB - 1 TB
Supervisor 3	i5-2400 3.10 GHz	10 GB - 500 GB
Supervisor 4	i5-2400 2.4 GHz	10 GB - 500 GB
Supervisor 5	i3-4030U 1.90 GHz	8 GB - 1 TB
Supervisor 6	i3-2330M 2.20 GHz	8 GB - 500 GB
Supervisor 7	Core 2 Duo E8600 3.33 GHz	8 GB - 1 TB

In addition, we conduct experiments using an Apache Storm cluster of 8 physical machines having hardware configurations as presented in Table 1.

Each machine runs Storm 0.9.7 on top of Ubuntu 10.4 with Java JDK 8u221, ZooKeeper 3.3.6, Zeromq 4.1.3, and the Java binding JZMQ in addition to other required Storm dependent libraries. A heterogeneous Storm cluster has one node running Nimbus daemon and ZooKeeper [47] with a relatively high storage capacity for log saving purposes. The other worker machines run supervisor daemon, each of which has a specific number of worker processes. Each worker process executes a subset of the topology, and each supervisor node has worker processes equal to the node's CPU cores.

We collect all the test results regarding the throughput and latency data from the Storm user interface (UI daemon). It is worth mentioning that the Storm system does not plot all the results and instead samples only 0.05% out of the total transactions to avoid overburdening the system. However, this aspect does not affect the average throughput because we run the test for 600 s, which represents adequate time for system stabilization and collecting sufficient samples to calculate the average throughput rates. In all the tests, the proposed algorithm assumes that a user preference exists in terms of the site location, and the cluster is distributed over at least two sites.
For our evaluation, the throughput of the overall topology (processed tuples per unit time) is limited by the performance bottleneck identified and minimized using the MT-Scheduler algorithm.

We use three commonly used microbenchmarks [26-28,38], namely, linear, diamond, and star topologies from [25], as shown in Fig. 7. The linear topology, as shown in Fig. 7(a), is the simplest structure and consists of 6 linear components. The diamond topology, as shown in Fig 7(b), includes five components, in which the spout feeds the middle three components, and the last bolt receives all the outgoing data. The star application, as shown in Fig. 7(c), is a multiple spout topology that transmits data tuples to the central bolt, which in turn transfers its processed tuples to the remaining components. For comparison, we evaluate our MT-Scheduler against the RR default scheduler and the state of the art adaptive scheduler [20].

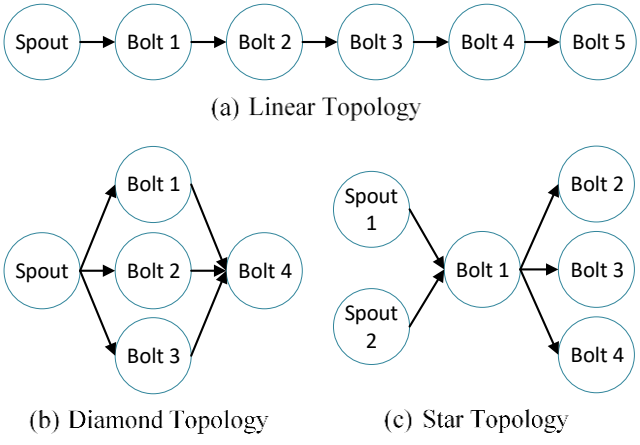

Figure 7. Test Topologies

The main goal of the proposed algorithm is to minimize the computational/communicational bottleneck time to achieve the maximum system throughput. Fig. 8 shows that the proposed algorithm outperforms both the default RR and the

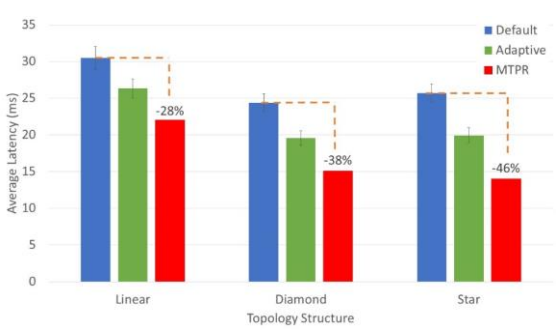

Figure 8 Total latency for different topologies

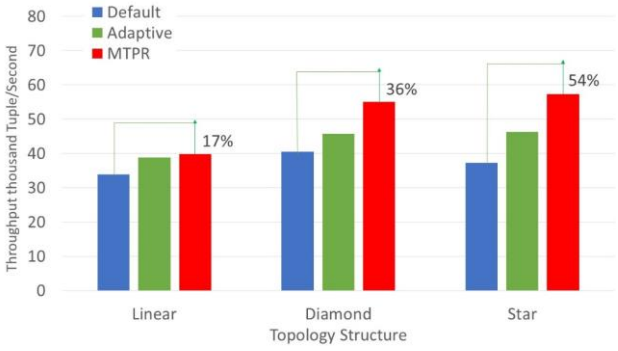

Figure 9 Throughput for different topologies

adaptive scheduler in terms of the latency (the elapsed time to ack a tuple after it is transmitted) under all the three topologies. Similarly, the average system throughput of the MT-Scheduler is higher than that of both the algorithms under all the three topologies, as shown in Fig. 9. The star topology, which has a complicated dependency structure, achieves the best performance, compared with the linear and diamond topologies.

7 Conclusions and Future Work

The proposed MT-Scheduler algorithm aims to maximize the system throughput for streaming applications in a Storm environment. The simulation evaluation results show the impact of the task complexity and data transfer rates on the scheduling performance. The proposed MT-Scheduler demonstrates satisfactory performance scalability when the cluster size scales up. Furthermore, we implement the MT-Scheduler in Apache Storm and use three microbenchmarks streaming topologies for testing and evaluation. The experimental results show that the MT-Scheduler outperforms both the default Storm RR scheduler and the adaptive scheduler. Compared with the default RR Storm scheduler, the MT-Scheduler reduces the system latencies by 28–46% and increases the throughput by 17–54%. We plan to implement the proposed MT-Scheduler in Apache Heron and test it on larger streaming applications in the Cloud environment. Furthermore, we intend to investigate the use of deep learning algorithms for dynamic workload balancing.

References

[1] M. Dias de Assunção, A. da Silva Veith, and R. Buyya, “Distributed data stream processing and edge computing: A survey on resource elasticity and future directions,” J. Netw. Comput. Appl., vol. 103, pp. 1–17, Feb. 2018.

[2] Q.-C. To, J. Soto, and V. Markl, “A survey of state management in big data processing systems,” *VLDB J.*, vol. 27, no. 6, pp. 847–872, Dec. 2018.

[3] F. A. Teixeira, F. M. Q. Pereira, H.-C. Wong, J. M. S. Nogueira, and L. B. Oliveira, “SIoT: Securing Internet of Things through distributed systems analysis,” *Future Gener. Comput. Syst.*, vol. 92, pp. 1172–1186, Mar. 2019.

[4] S. Imai, S. Patterson, and C. A. Varela, “Maximum Sustainable throughput Prediction for Data Stream Processing over Public Clouds,” in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017, pp. 504–513.

[5] S. Khan, K. A. Shakil, and M. Alam, “Cloud-Based Big Data Analytics—A Survey of Current Research and Future Directions,” in Big Data Analytics, vol. 654, V. B. Aggarwal, V. Bhatnagar, and D. K. Mishra, Eds. Singapore: Springer Singapore, 2018, pp. 595–604.

[6] S. Yi, C. Li, and Q. Li, “A Survey of Fog Computing: Concepts, Applications and Issues,” in Proceedings of the 2015 Workshop on Mobile Big Data - Mobidata ’15, Hangzhou, China, 2015, pp. 37–42.

[7] G. Jansen, I. Verbitskiy, T. Renner, and L. Thamsen, “Scheduling Stream Processing Tasks on Geo-Distributed Heterogeneous Resources,” in 2018 IEEE International Conference on Big Data (Big Data), Seattle, WA, USA, 2018, pp. 5159–5164.

[8] J. Xue, Z. Yang, S. Hou, and Y. Dai, “When computing meets heterogeneous cluster: Workload assignment in graph computation,” in 2015 IEEE International Conference on Big Data (Big Data), Santa Clara, CA, USA, 2015, pp. 154–163.

[9] W. A. Y. Aljoby, T. Z. J. Fu, and R. T. B. Ma, “Impacts of task placement and bandwidth allocation on stream analytics,” in 2017 IEEE 25th International Conference on Network Protocols (ICNP), Toronto, ON, 2017, pp. 1–

6.

[10] N. Kaur and S. K. Sood, “Dynamic resource allocation for big data streams based on data characteristics (5Vs),” Int. J. Netw. Manag., vol. 27, no. 4, p. e1978, Jul. 2017.

[11] M. Mortazavi-Dehkordi and K. Zamanifar, “Efficient resource scheduling for the analysis of Big Data streams,” Intell. Data Anal., vol. 23, no. 1, pp. 77–102, Feb. 2019.

[12] M.-A. Vasile, F. Pop, R.-I. Tutueanu, V. Cristea, and J. Ko\lodziej, “Resource-aware hybrid scheduling algorithm in heterogeneous distributed computing,” Future Gener. Comput. Syst., vol. 51, pp. 61–71, 2015.

[13] N. Tantalaki, S. Souravlas, and M. Roumeliotis, “A review on big data real-time stream processing and its scheduling techniques,” Int. J. Parallel Emergent Distrib. Syst., pp. 1–31, Mar. 2019.

[14] M.-A. Vasile, F. Pop, R.-I. Tutueanu, V. Cristea, and J. Ko\lodziej, “Resource-aware hybrid scheduling algorithm in heterogeneous distributed computing,” Future Gener. Comput. Syst., vol. 51, pp. 61–71, 2015.

[15] Z. Qian et al., “Timestream: Reliable stream computation in the cloud,” in Proceedings of the 8th ACM European Conference on Computer Systems, 2013, pp. 1–14.

[16] T. Akidau et al., “MillWheel: fault-tolerant stream processing at internet scale,” Proc. VLDB Endow., vol. 6, no. 11, pp. 1033–1044, 2013.

[17] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, “S4: Distributed stream computing platform,” in 2010 IEEE International Conference on Data Mining Workshops, 2010, pp. 170–177.

[18] M. Fu *et al.*, “Twitter Heron: Towards Extensible Streaming Engines,” in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, 2017, pp. 1165–1172.

[19] “Apache Storm.” [Online]. Available: https://Storm.apache.org/.

[20] L. Aniello, R. Baldoni, and L. Querzoni, “Adaptive [Online] scheduling in Storm,” in Proceedings of the 7th ACM international conference on Distributed event-based systems - DEBS ’13, Arlington, Texas, USA, 2013, p. 207.

[21] H. Röger and R. Mayer, “A Comprehensive Survey on Parallelization and Elasticity in Stream Processing,” ArXiv190109716 Cs, Jan. 2019.

[22] L. Sliwko, “A Taxonomy of Schedulers – Operating Systems, Clusters and Big Data Frameworks,” Glob. J. Comput. Sci. Technol., pp. 25–40, Mar. 2019.

[23] R. Mahmud, R. Kotagiri, and R. Buyya, “Fog Computing: A Taxonomy, Survey and Future Directions,” ArXiv161105539 Cs, pp. 103–130, 2018.

[24] J. Liu, E. Pacitti, and P. Valduriez, “A Survey of Scheduling Frameworks in Big Data Systems,” p. 28, 2018.

[25] M. Rychly, P. Koda, and P. Mr, “Scheduling Decisions in Stream Processing on Heterogeneous Clusters,” in 2014 Eighth International Conference on Complex, Intelligent and Software Intensive Systems, Birmingham, UK, 2014, pp. 614–619.

[26] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, “R-Storm: Resource-Aware Scheduling in Storm,” in Proceedings of the 16th Annual Middleware Conference on - Middleware ’15, Vancouver, BC, Canada, 2015, pp. 149–161.

[27] J. Xu, Z. Chen, J. Tang, and S. Su, “T-Storm: Traffic-aware [Online] scheduling in Storm,” in 2014 IEEE 34th International Conference on Distributed Computing Systems, 2014, pp. 535–544.

[28] T. Li, J. Tang, and J. Xu, “A predictive scheduling framework for fast and distributed stream data processing,” in 2015 IEEE International Conference on Big Data (Big Data), Santa Clara, CA, USA, 2015, pp. 333–338.

[29] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, “Optimal operator deployment and replication for elastic distributed data stream processing: Optimal Deployment and Replication for Elastic Data Stream Processing,” Concurr. Comput. Pract. Exp., vol. 30, no. 9, p. e4334, May 2018.

[30] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, “Optimal operator placement for distributed stream processing applications,” in Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems - DEBS ’16, Irvine, California, 2016, pp. 69–80.

[31] M. Nardelli, V. Cardellini, V. Grassi, and F. L. Presti, “Efficient Operator Placement for Distributed Data Stream Processing Applications,” IEEE Trans. Parallel Distrib. Syst., vol. 30, no. 8, pp. 1753–1767, Aug. 2019.

[32] M. Nardelli, “QoS-aware Deployment and Adaptation of Data Stream Processing Applications in Geo-distributed Environments,” Ph.D. Thesis, UNIVERSITY OF ROME TOR VERGATA, 2018.

[33] C. Li, J. Zhang, and Y. Luo, “Real-time scheduling based on optimized topology and communication traffic in distributed real-time computation platform of Storm,” J. Netw. Comput. Appl., vol. 87, pp. 100–115, Jun. 2017.

[34] W. Zhang, S. Li, L. Liu, Z. Jia, Y. Zhang, and D. Raychaudhuri, “Hetero-Edge: Orchestration of Real-time Vision Applications on Heterogeneous Edge Clouds,” in IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, Paris, France, 2019, pp. 1270–1278.

[35] S. Liu, J. Weng, J. H. Wang, C. An, Y. Zhou, and J. Wang, “An Adaptive [Online] Scheme for Scheduling and Resource Enforcement in Storm,” IEEE ACM Trans. Netw., pp. 1–14, 2019.

[36] A. Shukla and Y. Simmhan, “Model-driven scheduling for distributed stream processing systems,” J. Parallel Distrib. Comput., vol. 117, pp. 98–114, Jul. 2018.

[37] R. K. Kombi, N. Lumineau, P. Lamarre, N. Rivetti, and Y. Busnel, “DABS-Storm: A Data-Aware Approach for Elastic Stream Processing,” in Transactions on Large-Scale Data- and Knowledge-Centered Systems XL, vol. 11360, A. Hameurlain, R. Wagner, F. Morvan, and L. Tamine, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2019, pp. 58–93.

[38] L. Eskandari, J. Mair, Z. Huang, and D. Eyers, “T3-Scheduler: A topology and Traffic aware two-level Scheduler for stream processing systems in a heterogeneous cluster,” Future Gener. Comput. Syst., vol. 89, pp. 617–632, Dec. 2018.

[39] X. Liu and R. Buyya, “D-Storm: Dynamic Resource-Efficient Scheduling of Stream Processing Applications,” in 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS), Shenzhen, 2017, pp. 485–492.

[40] M. Caneill, A. El Rheddane, V. Leroy, and N. De Palma, “Locality-Aware Routing in Stateful Streaming Applications,” in Proceedings of the 17th International Middleware Conference on - Middleware ’16, Trento, Italy, 2016, pp. 1–13.

[41] “Apache Flink: Stateful Computations over Data Streams.” [Online]. Available: https://flink.apache.org/.

[42] “Apache SparkTM - Unified Analytics Engine for Big Data.” [Online]. Available: https://spark.apache.org/.

[43] M. Zhu, Q. Wu, N. S. V. Rao, and S. Iyengar, “Optimal pipeline decomposition and adaptive network mapping to support distributed remote visualization,” J. Parallel Distrib. Comput., vol. 67, no. 8, pp. 947–956, Aug. 2007.

[44] Q. Wu, M. Zhu, Y. Gu, and N. S. V. Rao, “System Design and Algorithmic Development for Computational Steering in Distributed Environments,” IEEE Trans. Parallel Distrib. Syst., vol. 21, no. 4, pp. 438–451, Apr. 2010.

[45] L. Blum, M. Shub, and S. Smale, “On a theory of computation over the real numbers; NP-completeness, recursive functions and universal machines,” in Proceedings 1988 29th Annual Symposium on Foundations of Computer Science, 1988, pp. 387–397.

[46] A. Al-Sinayyid and M. Zhu, “Maximizing The Processing Rate for Streaming Applications in Apache Storm,” in Proceedings of the 14th International Conference on Data Science (ICDATA’18), 2018.

[47] “Apache ZooKeeper.” [Online]. Available: https://zookeeper.apache.org/.

[48] “Amazon Timestream,” *Amazon Web Services, Inc.* [Online]. Available: https://aws.amazon.com/timestream/.

[49] “S4 Incubation Status - Apache Incubator.” [Online]. Available: http://incubator.apache.org/projects/s4.html.

The final publication is available at https://link.springer.com/article/10.1007/s11227-020-03223-z