Future Generation Computer Systems 112 (2020) 567–579

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Throughput optimization for Storm-based processing of stream data on clouds

Huiyan Cao [a], Chase Q. Wu [a,*], Liang Bao [b], Aiqin Hou [c], Wei Shen [d]

[a] Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102, USA
[b] School of Software, XiDian University, Xi'an, ShaanXi 710071, China
[c] School of Information Science and Technology, Northwest University, Xi'an, Shaanxi 710127, China
[d] School of Informatics Science and Technology, Zhejiang Sci-Tech University, Hangzhou, Zhejiang, 310018, China

ARTICLE INFO

Article history:
Received 19 May 2019
Received in revised form 14 April 2020
Accepted 6 June 2020
Available online 10 June 2020

Keywords:
Scientific workflows
Apache Storm
Workflow mapping
Throughput optimization

ABSTRACT

There is a rapidly growing need for processing large volumes of streaming data in real time in various big data applications. As one of the most commonly used systems for streaming data processing, Apache Storm provides a workflow-based mechanism to execute directed acyclic graph (DAG)-structured topologies. With the expansion of cloud infrastructures around the globe and the economic benefits of cloud-based computing and storage services, many such Storm workflows have been shifted or are in active transition to clouds. However, modeling the behavior of streaming data processing and improving its performance in clouds still remain largely unexplored. We construct rigorous cost models to analyze the throughput dynamics of Storm workflows and formulate a budget-constrained topology mapping problem to maximize Storm workflow throughput in clouds. We show this problem to be NP-complete and design a heuristic solution that takes into consideration not only the selection of virtual machine type but also the degree of parallelism for each task (spout/bolt) in the topology. The performance superiority of the proposed mapping solution is illustrated through extensive simulations and further verified by real-life workflow experiments deployed in public clouds in comparison with the default Storm and other existing methods.

1. Introduction

Extreme-scale applications in various scientific, industry, and engineering domains have been the major source of big data. In many of these applications, there is an increasing need to process and analyze datasets as they are generated and transferred in real time for various purposes such as stock prediction, malfunction detection, social network analysis, and log data processing. To meet such demands, a wide range of computing engines have been developed and deployed for streaming data processing, including Apache Storm [1], Apache Flink [2], Apache Spark (Spark Streaming) [3], Apache Samza [4], Apache Apex [5], and Google Cloud Dataflow [6]. For example, Yahoo adopted Apache Storm to replace the internally developed S4 platform [7]; JStorm [8], now being merged into Apache Storm, and Heron [9] are heavily used by Alibaba Inc. and Twitter Inc., respectively; Spark Streaming and Flink are also gaining a widespread adoption in industry. In fact, real-time streaming data processing systems have become an indispensable building block in the entire big data ecosystem.

As one of the most commonly used systems for streaming data processing, Apache Storm provides a workflow-based mechanism to execute directed acyclic graph (DAG)-structured topologies[1]. In recent years, we have witnessed a rapid deployment of cloud infrastructures around the globe and great economic benefits brought by cloud-based computing and storage services. As a result, many such Storm workflows have been shifted or are in active transition to cloud environments. As most public clouds adopt a pay-as-you-go service model, one additional constraint on financial budget must be considered in addition to traditional performance optimization goals. However, efforts in improving the performance of streaming data processing in clouds are still very limited.

In this paper, we construct rigorous mathematical models to analyze both the time and financial cost of Storm-based workflow execution and formulate a Storm Topology Mapping problem for maximum throughput in clouds under Budget Constraint, referred to as STM-BC. We show STM-BC to be NP-complete and design a heuristic solution that takes into consideration the parallelism of each task (spout/bolt) in the topology. The performance superiority of the proposed mapping solution is illustrated

* Corresponding author.
E-mail addresses: hc334@njit.edu (H. Cao), chase.wu@njit.edu (C.Q. Wu), baoliang@mail.xidian.edu.cn (L. Bao), houaiqin@nwu.edu.cn (A. Hou), shenwei@zstu.edu.cn (W. Shen).

[1] The workflow structure in Storm is referred to as a topology, and hereafter, these two terms are used interchangeably.

https://doi.org/10.1016/j.future.2020.06.009
0167-739X/

through extensive simulations and further verified by real-life workflow experiments deployed in public clouds in comparison with the default Storm and other existing methods. Our workflow mapping solution offers IaaS providers a cost-effective resource allocation scheme to meet the budget constraint specified by the user, and meanwhile also serves as a cloud resource provisioning reference for scientific users to make proactive and informative resource requests.

The rest of the paper is organized as follows. Section 2 describes related work on stream data processing. Section 3 models the performance of Storm workflows on clouds and formulates STM-BC. Section 4 designs a workflow mapping algorithm. Sections 5, 6 and 6.3 present simulation results, two sets of experimental results for evaluation, and performance analysis, respectively. Section 7 concludes our work.

2. Related work

We conduct a survey of related work on streaming data processing in various computing environments.

Many existing efforts have been focused on workflow mapping or job scheduling in grid environments under different mapping and resource constraints. Agarwalla et al. proposed *Streamline*, a workflow scheduling scheme for streaming data, which places a coarse-grain dataflow graph on available grid resources [10]. Similar mapping problems are also studied in the context of sensor networks. Sekhar et al. proposed an optimal algorithm for mapping subtasks onto a large number of sensor nodes based on an A^* algorithm [11].

More recently, as Storm gains its popularity for streaming data processing in big data systems, a number of improvements have been made to Storm in either physical or cloud-based clusters. The current Storm platform employs a pseudo-random round-robin task scheduling and placement scheme without considering resource availability in the underlying cluster. This default scheme is simple but does not always yield the best performance in terms of workflow throughput and resource utilization. Many efforts have been made to improve throughput performance using resource-aware scheduling. In [12], Peng et al. proposed R-Storm (Resource-Aware Storm), a system that implements resource-aware scheduling within Storm, which is designed to increase overall throughput by maximizing resource utilization while minimizing network latency. In [13], Eskandari et al. considered the data transfer rate and traffic pattern between Storm's tasks and assign task pairs with heavy communication to the same node by dynamically employing two phases of graph partitioning. In [14], Chen et al. presented the design, implementation, and evaluation of G-Storm, a GPU-enabled parallel system based on Storm, which harnesses the massively parallel computing power of GPUs for high-throughput stream data processing.

There also exists some work on Storm scheduling that considers various application features such as data transfer, workflow topology, and QoS. In [15], Xu et al. proposed T-Storm, a traffic-aware online scheduler in Storm, to minimize inter-node and inter-process traffic for better performance with even fewer worker nodes. In [16], Aniello et al. proposed two schedulers for Storm to improve performance by adapting the deployment to application topologies, and by rescheduling the deployment at runtime based on traffic information. In [17], Cardellini et al. extended the Storm architecture by designing and implementing the support for distributed QoS-aware scheduling and run-time adaptivity.

Different from the aforementioned work that considers resources or application features, we take an orthogonal approach to maximize the throughput performance of Storm workflows in clouds by deciding an appropriate degree of parallelism for each component task of the topology and selecting a suitable virtual machine (VM) type for each component task processing an input instance.

3. Cost models and problem formulation

We begin with the construction of cost models used for problem definition. We consider a Storm topology as a directed acyclic graph (DAG) $G_{tp}(V_{tp}, E_{tp})$ with $|V_{tp}|$ modules[2] and $|E_{tp}|$ edges, each of which represents the execution dependency and data movement between two neighbor modules. A Storm-based streaming application is executed in a heterogeneous cluster deployed in a cloud with n virtual machine (VM) types $VT = \{vt_1, vt_2, \ldots, vt_n\}$, for each, there may exist multiple VM instances. Each VM type vt has a set of performance attributes including CPU frequency $f_{CPU}(vt)$, number of virtual cores $nc(vt)$, and memory capacity $m(vt)$, as well as a commonly used "pay-as-you-go" VT pricing model $p(vt) = f(f_{CPU}(vt), nc(vt), m(vt))$, which determines the financial cost per time unit for using a VM instance of that type. Note that the actual computing or processing power of a given core is typically measured in unit of MIPS (million instructions per second). In this work, we consider a single cloud environment and the cost for data transfer is not accounted as in most real-life public clouds.

We define a Storm topology mapping scheme $\mathscr{M}$ as

$$\mathscr{M} : v_{tp}(DoP) \rightarrow VM(vt), \text{ for all } v_{tp} \in V_{tp}, \tag{1}$$

where $VM(vt)$ represents the VT selection and DoP represents the degree of parallelism for module v_{tp}, which denotes a spout or a bolt *Bolt* in the Storm topology.

In Storm [1], a data stream is comprised of tuples. As a source of data streams in a topology, a spout reads tuples from an external source and emits them into the topology. A bolt represents a data processing unit in the topology, such as filtering, aggregation, join, communicating with databases, etc. Based on a pre-specified DoP, each spout or bolt executes multiple tasks concurrently across the cluster, each of which corresponds to one thread of execution, and stream groupings define how to send tuples from one set of tasks to another. Note that DoP determines the number of VM instances of vt selected for executing v_{tp}. In other words, for each v_{tp}, we create a number DoP of worker instances that are launched on different VMs. In this work, a single worker is created on each VM instance with a single executor to process one tuple of input data.

We define the gap time of module v_{tp} as the time interval between the finish time of two adjacent tuples processed by two copies of v_{tp}. Note that the gap time of each module may or may not be uniform during the entire streaming data processing. We have the following theorem on the pattern of the module gap time.

Theorem 1. *The gap time of any module v_{tp} in the Storm topology occurs periodically.*

Proof. We prove Theorem 1 by mathematical induction. In the base case, we analyze the first bolt $Bolt_1$ with $N_1 = m$ copies running in parallel. As shown in Fig. 1, the gap time (i.e., t_1, t_2, etc.) between m workers of the first bolt in the topology is the same as the time interval of two adjacent tuples emitted from

[2] We refer to the smallest computing entity in a general workflow (or more specifically, spout/bolt in a Storm topology) as a computing module, which represents either a serial computing task or a parallel processing job such as a typical MapReduce program in Hadoop.

the spout. T_1 denotes the execution time of one copy of $Bolt_1$. Obviously, the base case is established.

Suppose that $Bolt_i$'s gap time occurs periodically. Basically, there are three cases of $Bolt_{i+1}$:

- Case 1: when $N_{i+1} = N_i$, i.e., $Bolt_i$ and $Bolt_{i+1}$ have the same number of workers. Let T_i denote the execution time of a worker of $Bolt_i$ processing one tuple. Fig. 2 shows the case when $T_i = T_{i+1}$, so for $Bolt_i$, worker j of $Bolt_{i+1}$ always has a delay of T_i after worker j of $Bolt_i$, where $j = 1, 2, \ldots, N_i$. Hence, $Bolt_{i+1}$ has the same cyclic pattern as $Bolt_i$. Fig. 3 shows the case when $T_i > T_{i+1}$. Similar to Fig. 2, there is still a delay of T_i between $Bolt_i$ and $Bolt_{i+1}$ on each corresponding worker, which means that $Bolt_{i+1}$ has the same cyclic pattern as $Bolt_i$. Fig. 4 shows the case when $T_i < T_{i+1}$. Each corresponding worker has a delay of T_{i+1}. Since $T_i < T_{i+1}$, the gap time is different from that of $Bolt_i$. However, there is a one-to-one mapping between the finish time of $Bolt_i$ and $Bolt_{i+1}$, as well as the gap time of $Bolt_i$ and $Bolt_{i+1}$. Therefore, $Bolt_{i+1}$'s gap time sequence can be mapped to $Bolt_i$'s gap time sequence, and $Bolt_{i+1}$ should have the same cyclic pattern as $Bolt_i$.
- Case 2 when $N_i > N_{i+1}$. Fig. 5(a) shows the case when $T_i < T_{i+1}$. We assign the kth worker of $Bolt_{i+1}$ to process the next tuple emitted from the jth worker of $Bolt_i$. There is a one-to-one mapping from the finish time of each tuple processed by $Bolt_i$ to the finish time of the same tuple processed by $Bolt_{i+1}$. Since the gap time of $Bolt_i$ has a cyclic pattern, so does the gap time of $Bolt_{i+1}$. Fig. 5(b) shows the case when $T_i > T_{i+1}$, where the situation is similar to Fig. 5(a). The only difference is that there may exist a certain waiting time between the first tuple's start time in the next cycle and the last tuple's finish time of each worker of $Bolt_{i+1}$. Since the tuple mapping and the corresponding delay time remain the same in each cycle, so the cyclic pattern carries on in $Bolt_{i+1}$. When $T_i = T_{i+1}$, it is obvious that $Bolt_{i+1}$ exhibits a cyclic pattern.
- Case 3 when $N_i < N_{i+1}$. The execution dynamics analysis is similar to Case 2 and hence is omitted.

Note that $Bolt_i$ may have multiple upstream bolts. Assume that there are n upstream bolts $Bolt_k$, where $k = i-n, \ldots, i-2, i-1$. Since the number of workers for each bolt may be different, we consider the lowest common multiple LCM_i of all N_k as the number of workers for each bolt. These n upstream bolts can be treated as a single virtual bolt with LCM_i workers. The jth worker, $j = 1, 2, \ldots, LCM_i$, emits a tuple at the latest time when $Bolt_k$ emits the jth tuple. After the transformation, based on the above case, we can prove that the gap time of any bolt has a cyclic pattern.

Proof ends.

This cyclic pattern is critical to modeling the throughput of any module v_{tp}, which denotes either a spout or a bolt *Bolt* in the Storm topology. According to Theorem 1, we plot the relationship between tuple index and processing time for each tuple on module v_{tp} in Fig. 6, which shows two cycles for illustration. To calculate throughput, we consider a period of time and the number of tuples processed during this period. Since the gap time of $Bolt_i$ has a cyclic pattern, we calculate the throughput by counting the number of tuples processed per cycle. The first cycle is from time 0 to n and the second one is from time $n+1$ to $2n$, where n is the end time of the first cycle in ms (time unit). Hence, the cycle time $CT_{v_{tp}} = n$. In each cycle, v_{tp} processes m tuples, defined as tuple count per cycle $TCPC_{v_{tp}}$. We define the throughput $T(\mathscr{M}, v_{tp})$ of module v_{tp} under the mapping scheme

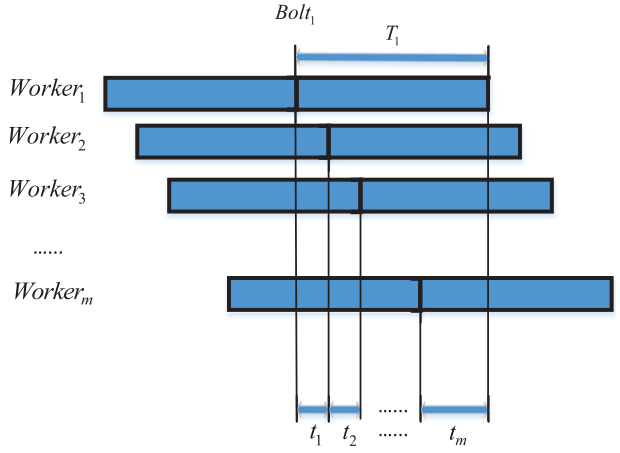


Fig. 1. Execution dynamics of the first bolt of the topology with DoP = m, i.e., there are m concurrent workers executing the first bolt.

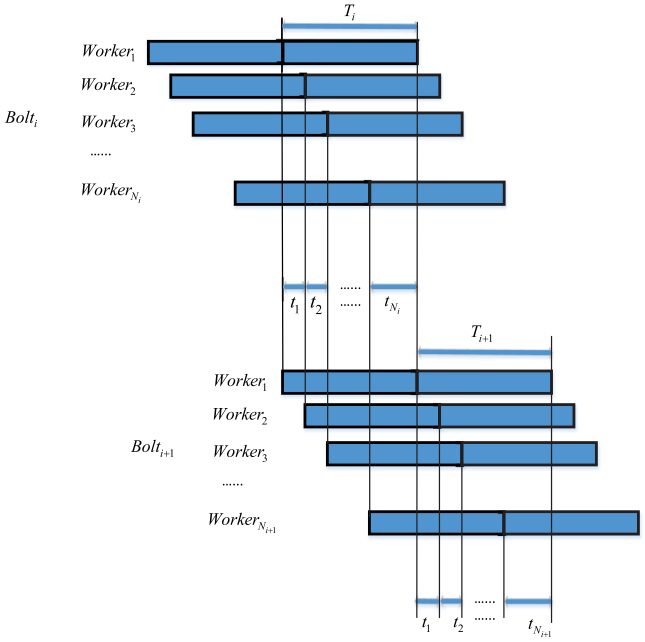


Fig. 2. Execution dynamics in Case 1: the gap time when $T_i = T_{i+1}$.

of $\mathscr{M}$ as the inverse of the average processing time for each tuple during each cycle:

$$T(\mathscr{M}, v_{tp}) = \frac{1}{\frac{CT_{v_{tp}}}{TCPC_{v_{tp}}}} = \frac{TCPC_{v_{tp}}}{CT_{v_{tp}}}. \tag{2}$$

A bottleneck is a process in a chain of processes whose computing power limits the computing capacity of the whole execution chain, and may result in stalls in execution. A global bottleneck module is the one with the smallest $T(\mathscr{M}, v_{tp})$, and the throughput of the entire topology is determined by the bottleneck module's throughput, defined as:

$$GT(\mathscr{M}) = \min_{v_{tp} \in V_{tp}} T(\mathscr{M}, v_{tp}). \tag{3}$$

Based on the above mathematical models, we formulate a Storm Topology Mapping problem for maximum throughput in clouds under Budget Constraint, referred to as STM-BC, as follows.

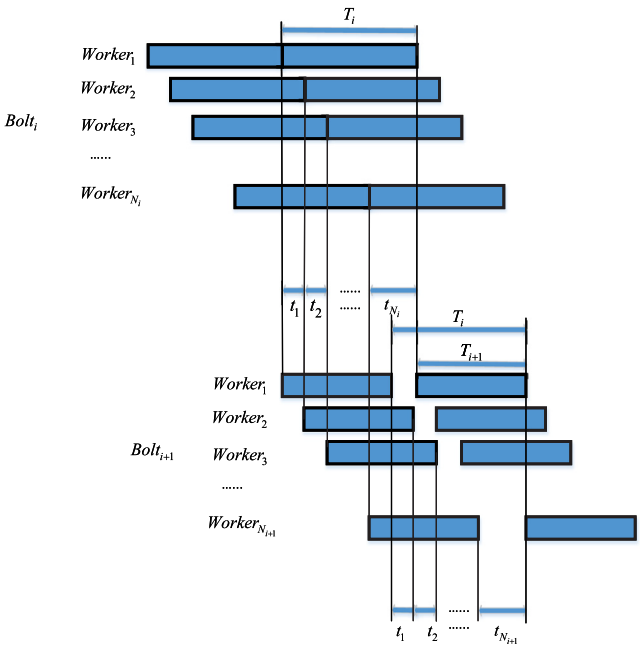


Fig. 3. Execution dynamics in Case 1: the gap time when $T_i > T_{i+1}$.

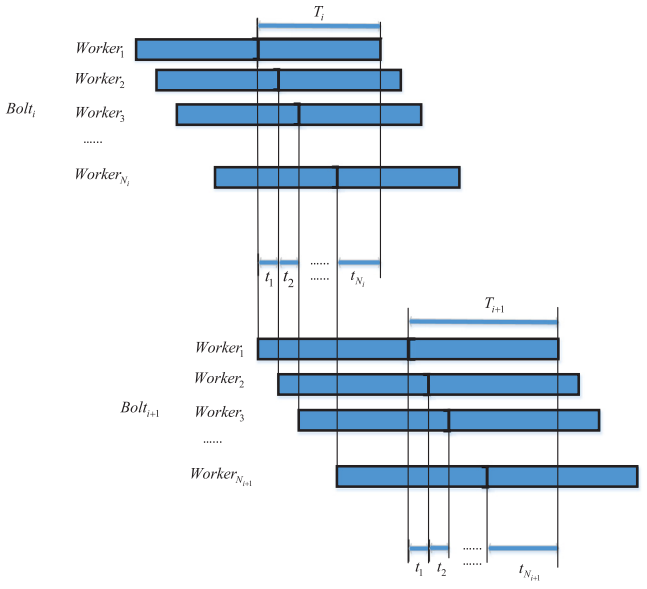


Fig. 4. Execution dynamics in Case 1: the gap time when $T_i < T_{i+1}$.

Definition 1. Given a DAG-structured Storm topology $G_{tp}(V_{tp}, E_{tp})$, a set VT of available VM types, and a fixed financial budget b per time unit, we wish to find a topology mapping scheme $\mathscr{M}$ to achieve the Maximum Throughput (MT):

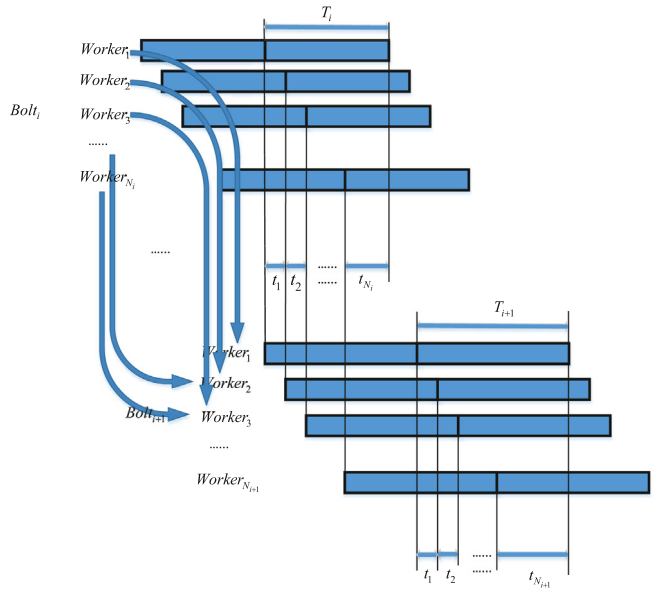
$$MT = \max_{\text{all possible } \mathscr{M}} MT(\mathscr{M}), \tag{4}$$

while satisfying the following budget constraint:

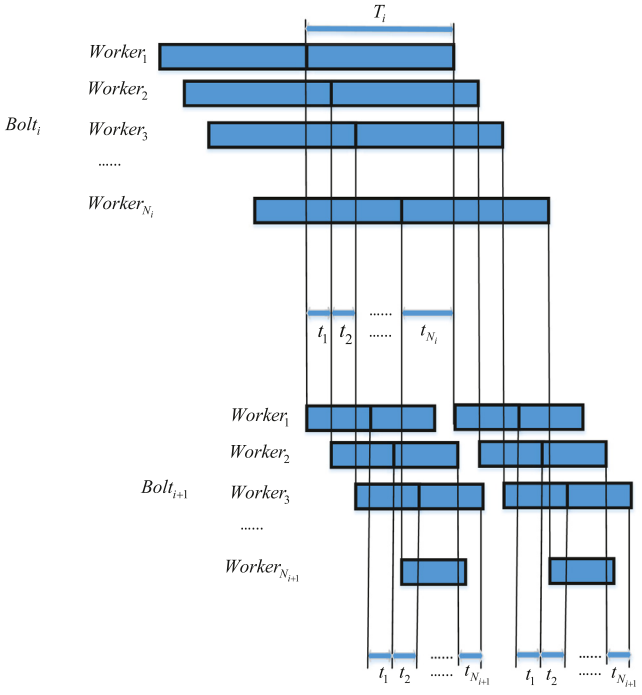
$$C \leq b, \tag{5}$$

where C is the total financial cost of VMs used for the Storm topology execution per time unit, calculated as

$$C = \sum_{VMs(vt) \text{ used in } \mathscr{M}} p(vt), \tag{6}$$



(a) The gap time when $T_i < T_{i+1}$.



(b) The gap time when $T_i > T_{i+1}$.

Fig. 5. Execution dynamics in Case 2.

where v_{tp} is mapped to $VM(vt)$, for each $v_{tp} \in V_{tp}$.

The problem formulated above is a generalized version of the MFR-ANR problem in [18], which only considers a pipeline structured workflow without parallel computing for each module. Specifically, in MFR-ANR, the authors consider a linear computing pipeline consisting of a number of sequential modules and a computer network represented as a directed arbitrary graph. They aim to find a one-on-one mapping scheme between a module and a computing node to achieve maximum frame rate. Note

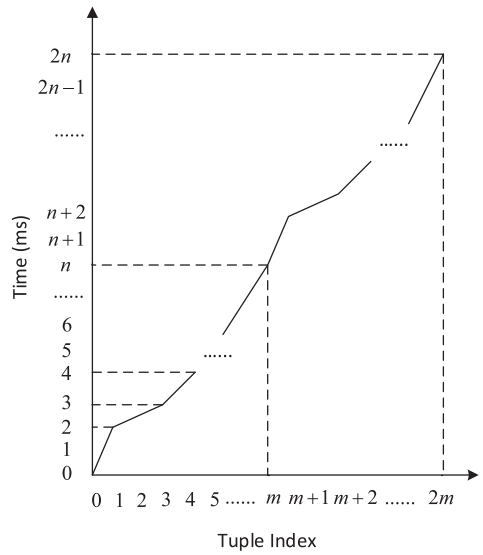


Fig. 6. Illustration of gap time for throughput calculation.

that a pipeline is a special case of workflow, and one-on-one mapping does not allow parallel computing as one module must be processed exclusively by one computing node. In our work, we formulate STM-BC, which supports parallel computing since one module (either a spout or a bolt in Storm) can be processed by multiple workers. Since MFR-ANR, which is a special case of STM-BC, has been proved to be NP-complete and non-approximable, so is STM-BC. Hence, we focus on the design of heuristic solutions to our problem.

We would like to point out that our cost models can be adapted to other stream data processing platforms, such as Spark Streaming [3] workflows where each module in the workflow is a Spark Streaming job. Such adapted cost models can be used to find the mapping of Spark jobs in the workflow to a set of physical or virtual computing nodes.

4. Algorithm design

We design a bottleneck-oriented topology mapping (BOTM) algorithm in Storm to solve STM-BC. BOTM determines not only the VT selection but also the degree of parallelism (DoP) for each module in the topology. The key idea is to iteratively identify the global bottleneck module and strategically compute an appropriate adjustment for this module's VT selection and degree of parallelism to achieve the maximum increase of the global workflow throughput. Note that the default scheduler in Storm assigns executors in a round-robin manner without considering the global bottleneck.

4.1. Bottleneck-oriented topology mapping

The pseudocode of BOTM is provided in Alg. 1, which consists of the following key steps.

Step (1) Sort the available VM types VT according to the total CPU frequency of all virtual cores, which determines the aggregate computing power in unit of MIPS (million instructions per second), memory space, and I/O speed. Initially, every module in the workflow is assigned to the worst vt in the cloud. If this mapping scheme exceeds the budget, there is no feasible solution; otherwise, continue.

Step (2) Calculate the throughput for each module in the workflow based on the initial mapping scheme from Step 1. The module with the smallest throughput determines the global bottleneck.

Step (3) Call Function $SelectVT()$ in Alg. 2 check if it is possible to adjust the degree of parallelism and upgrade the type of VMs in order to achieve a higher global throughput within the budget. There are multiple options to determine the degree of parallelism and the vt for the global bottleneck module: add one more VM of the current vt within N_{vt}, which denotes the number of VM instances of the current vt; try to upgrade vt one level up at a time until reaching the best vt, and for each vt, gradually decrease the degree of parallelism from the degree of the current vt selection to 1. Every time we try to make an adjustment, we first eliminate the options that exceed the budget, and then compare the new global throughput after the adjustment. The option that results in the maximum increase in the global throughput is selected. Note that after each adjustment, the global bottleneck module may change.

Step (4) If any upgrade adjustment within the budget does not lead to a better global throughput, the algorithm terminates. Otherwise, update the degree of parallelism and the vt for the current bottleneck module, as well as the current global throughput.

Step (5) Go back to Step 2, and repeat the above process until no feasible upgrade adjustment option is available.

Algorithm 1: BOTM

Input: a DAG-structured topology $G_{tp}(V_{tp}, E_{tp})$, a set VT of VM types, the number N_{vt} of available VM instances of each vt, and a fixed financial budget b.

Output: the max throughput MT of the topology.

```
1: curTH = 0;
2: MT = 0;
3: sort the VM type VT in an increasing order of system resources;
4: Assign every module v_tp ∈ V_tp to a VM instance of the worst vt for the
   topology;
5: if the cost > b then
6:    throw ERROR("budget insufficient.");
7: Calculate curTH and assign MT with curTH
8: while true do
9:    tIndex = the index of the bottleneck module with the smallest throughput
      curTH;
10:   {tType, tNum} = selectVT(tIndex, VT, N_vt, b, curTH);
11:   if (tType == −1) then
12:      break;
13:   update tType and tNum for this bottleneck module;
14:   MT = curTH;
15: return MT.
```

After identifying the global bottleneck module in Step 2, we try to increase the throughput of the current global bottleneck module by making an adjustment to the VT selection and the degree of parallelism of this module within the budget in Step 3. We consider several adjustment options and select one that leads to the maximum increase of the global workflow throughput. We would like to point out that the global bottleneck may shift to a different module after the adjustment, and therefore, it does not always yield the best performance if we only maximize the throughput increase of the current bottleneck module.

To increase the throughput of the current bottleneck module, there are two ways to make adjustments: (i) increase the number of VM instances of the current vt by one; (ii) choose a more powerful vt and vary the module's DoP from its current DoP to one. Any option that exceeds the budget constraint is ruled out. Among the feasible options within the budget constraint, we select the option that maximizes the throughput increase of the

Algorithm 2: SelectVT

Input: the index of the bottleneck module *tIndex*, the VM type *VT* with the available number N_{vt} of VM instances of each vt, a fixed financial budget b, and the current topology throughput *curTH*.

Output: the VT type *tType* and the degree *tNum* of parallelism for the bottleneck module.

```
tType = −1;
tNum = −1;
curType = VT type of tIndex;
curNum = the degree of parallelism for the bottleneck module of tIndex;
for all vt ∈ VT do
  if vt is the same as curType and one more VM instance of vt is available then
    assign one more VM instance of vt to module of tIndex;
    calculate the topology throughput TH after the adjustment;
    if cost ≤ b and TH > curTH then
      tType = vt;
      curTH = TH;
      tNum = curNum + 1;
  else if vt is better than curType then
    tmpNum = curNum;
    while tmpNum > 0 do
      assign tmpNum VM instances of vt to module of tIndex;
      calculate the topology throughput TH after the adjustment;
      if cost ≤ b and TH > curTH then
        tType = vt;
        curTH = TH;
        tNum = tmpNum;
      tmpNum − −;
return {tType, tNum}.
```

entire workflow. Note that in some cases adding resources may result in reduced cost. Hence, we calculate the financial cost every time when we make a change to the selection of virtual machines.

In Storm, users are allowed to change the *DoP* for each module of the topology, but it is generally difficult for them to decide the most suitable *DoP* for each module. In many cases, users may specify an arbitrary *DoP* based on their empirical study. Our work not only selects the suitable vt but also determines the suitable degree of parallelism for each module of the workflow.

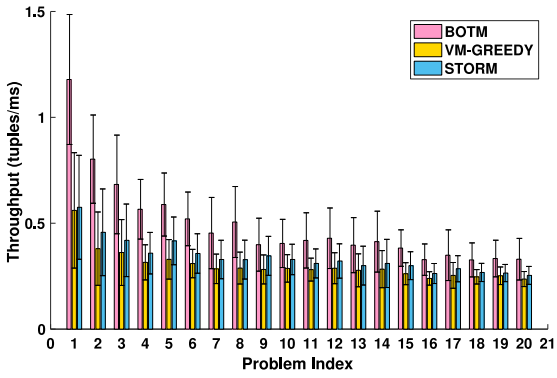
The time complexity of BOTM is $O(|V_{tp}|\cdot\max(N_{vt})\cdot|VT|)$, where $|VT|$ is the number of VM types, $\max(N_{vt})$ denotes the largest number of VM instances for all $vt \in VT$.

5. Simulation-based performance evaluation

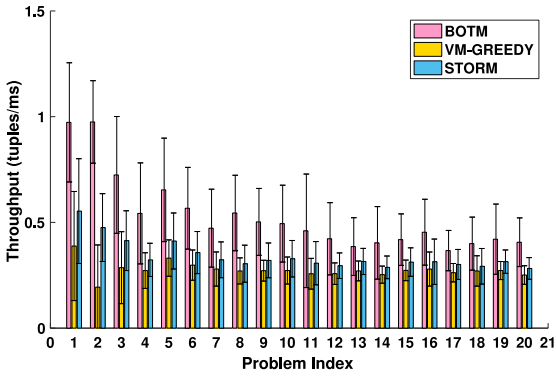
5.1. Simulation settings

We implement the proposed BOTM algorithm in C++ and evaluate its performance in comparison with the default Storm configuration, denoted as STORM_DEFAULT, and a heuristic algorithm VM-GREEDY. By comparing with Storm's default scheduler, which is used by many real-life applications, we are able to examine the benefits of BOTM to both service providers and end users when executing budget-constrained workflows. VM-GREEDY is a commonly used benchmark method that takes a greedy strategy for VM optimization to assign as many high-end VM instances as possible within the budget. The source code of BOTM implementation is available for download in GitHub Repository [19].

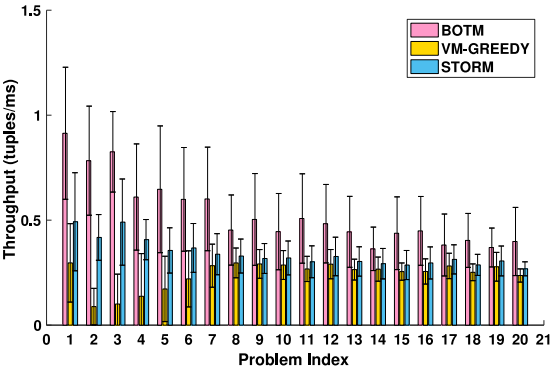
The problem size (reflected by the problem index) is defined as a 2-tuple $(|V_{tp}|, |E_{tp}|)$, where $|V_{tp}|$ is the number of Storm topology tasks, and $|E_{tp}|$ is the number of topology links. We generate topology instances of different scales in a random manner as follows [20]: (i) lay out all $|V_{tp}|$ modules sequentially along a pipeline, each of which is assigned a workload randomly generated within the range $[5, 500]$, which represents the total number of million instructions; (ii) for each module, add an input edge from a randomly selected preceding module and add an output edge to a randomly selected succeeding module (the first spout module only needs output and the last bolt module only needs input); (iii) randomly select two modules from the pipeline and add a directed edge between them (from left to right) until reaching the given number of edges.



(a) The average throughput with standard deviations across 400 instances (20 different problem sizes × 20 random workflow instances) at budget level 1.



(b) The average throughput with standard deviations across 400 instances (20 different problem sizes × 20 random workflow instances) at budget level 3.



(c) The average throughput with standard deviations across 400 instances (20 different problem sizes × 20 random workflow instances) at budget level 5.

Fig. 7. Performance measurements for simulations under different budget levels.

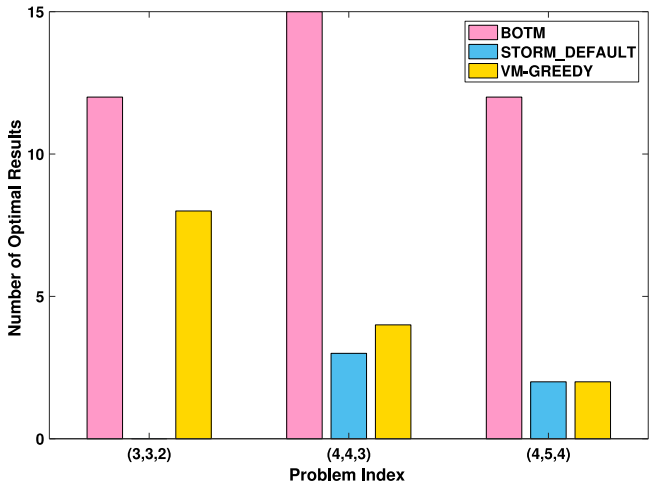


Fig. 8. The number of optimal results among 50 instances (10 workflow instances × 5 budget levels) produced by BOTM, VM-GREEDY and STORM_DEFAULT, respectively, under different problem sizes.

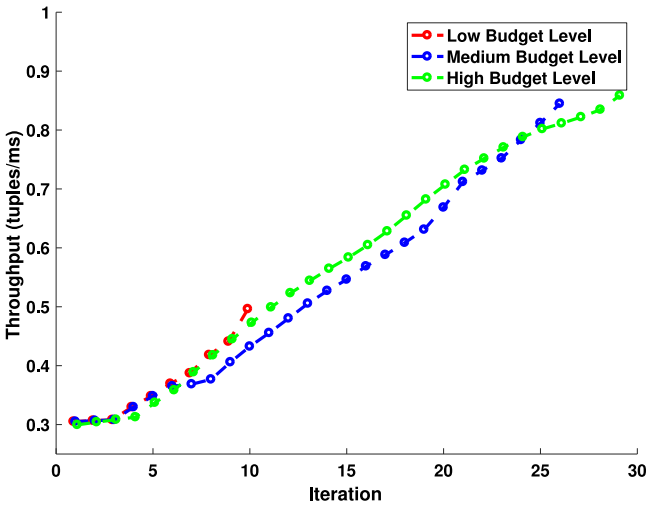


Fig. 9. The optimization process of BOTM running the problem instance of Index 5 in the simulations under three different budget levels.

We compare BOTM with the other two algorithms in comparison in terms of workflow throughput under the same budget constraint.

5.1.1. Comparison with optimal solutions

We compare BOTM with optimal solutions in three small-scale problems of (3, 3, 2), (4, 4, 3), and (4, 5, 4), each in the form of (number of modules, number of edges, number of VTs). For each problem size, we randomly generate 10 problem instances with different module workloads and DAG topologies. In each problem instance, we specify five different budget levels. We run all three algorithms on these instances and compare the throughput measurements with the optimal ones computed by an exhaustive search-based approach. Fig. 8 shows the number of optimal results among 50 instances (10 workflow instances × 5 budget levels) achieved by BOTM, VM-GREEDY, and STORM_DEFAULT, respectively, under different problem sizes. In (3, 3, 2), STORM_DEFAULT does not produce any optimal solution and thus is not visible in the chart. We observe that BOTM is more likely to achieve the optimality than the others in a statistical sense, which indicates the efficacy of BOTM. However, since these are small-scale problem instances, the absolute values of the differences from the optimal results are not significant.

5.2. Comparison with other methods

In the simulation, we consider 16 virtual machine types with their respective system specifications and in-cloud financial costs randomly selected from a range corresponding to commonly used virtual machines provisioned by Amazon Web Services (AWS) [21]. We consider 20 problem sizes from small to large scales, indexed from 1 to 20. For each problem size, we randomly generate 20 problem instances, in each of which, we choose 6 budget levels with an equal interval of $\Delta b = (b_{max} - b_{min})/6$ within a certain budget range $[b_{min}, b_{max}]$, where B_{min} is 10% more than the minimum budget to run the entire workflow on the worst cluster, and B_{max} is 10% more than the maximum budget to run the entire workflow on the best cluster. For each of the 6 budget levels from low to high levels, indexed from 1 to 6, we run the scheduling simulation by iterating through 20 problem sizes from small to large scales. We measure the average throughput with a standard deviation achieved by BOTM, VM-GREEDY, and STORM_DEFAULT, respectively. These measurements show the performance superiority of BOTM at each of the six budget levels. The results at levels 1, 3 and 5 are plotted in Fig. 7 for a visual comparison.

These performance results show that BOTM achieves performance improvement over VM-GREEDY and STORM_DEFAULT. Such performance improvements are considered significant for stream data processing in large-scale scientific applications. On average, the simulation results show that BOTM achieves a throughput that is 2.3 times of VM-GREEDY and 50% higher than STORM_DEFAULT. This is considered to be a significant improvement when dealing with large-scale stream data.

5.3. Convergence of BOTM

To investigate the convergence property of BOTM, we run this algorithm on the problem instance of Index 5 under three different budget levels, i.e., low, medium, and high. The low budget level is 10% more than the budget that is sufficient for the workflow to be executed using the worst virtual machines; the high budget level is 10% less than the budget that is sufficient for the workflow to be executed using the best virtual machines; the medium budget level is 50% of the budget that is sufficient for the workflow to be executed using the best virtual machines. We plot the optimization process of BOTM in these three scenarios in Fig. 9, which shows that BOTM converges to the maximum throughput after 30 iterations within less than one second. For problem index 10 and above, we observe that BOTM converges after at most 50 iterations.

6. Experiment-based performance evaluation

In this section, we conduct two sets of experiments on two real-life datasets. Different data volumes (12 GB and less than 1 GB data) are tested for scalability evaluation.

6.1. Experiment 1 with flight data

6.1.1. Storm topology

We conduct Storm experiments for streaming data processing to compute various statistics on 22 years of global flight datasets of about 12 GB from 1987 to 2008 at Statistical Computing [22]. The topology structure is shown in Fig. 10, where every module is a task (spout/bolt): w_0 emits streaming data instances every 1 ms;

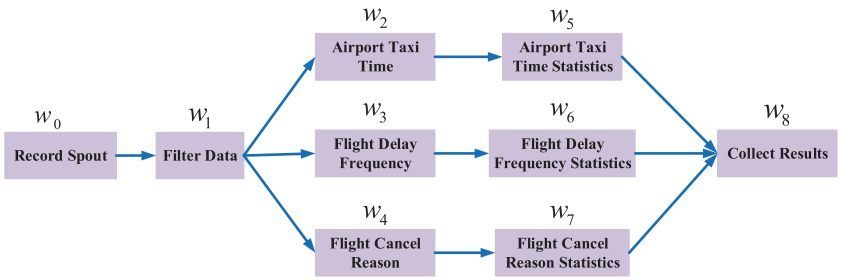


Fig. 10. The structure of the Storm topology for flight data processing.

Table 1
System specifications of different VM types in the experiment.

VM type	Instance name	Availability zone	CPU (GHz)	RAM (GB)	Num of instances	Price ($/min)
$vt1$	*t2.small*	US West (Oregon)	2.5×1	2	9	0.0230
$vt2$	*t2.medium*	US West (Oregon)	2.5×2	4	4	0.0464
$vt3$	*t2.xlarge*	US West (Oregon)	2.4×4	16	4	0.1856
$vt4$	*t2.2xlarge*	US West (Oregon)	2.4×8	32	4	0.3712

Table 2
Execution time matrix T_e in *ms*.

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8
$vt1$	1.85	46.21	53.44	105.83	97.19	12.56	17.39	55.70	11.23
$vt2$	1.47	26.79	6.57	12.04	39.98	1.09	4.18	3.52	2.05
$vt3$	1.38	14.58	5.63	11.06	26.18	1.08	4.09	3.47	1.54
$vt4$	1.21	13.33	3.13	6.42	24.77	1.07	3.37	2.62	1.41

Table 3
The VM instances of the Storm cluster provisioned under different mapping schemes in AWS.

Cluster	$vt1$ Inst.	$vt2$ Inst.	$vt3$ Inst.	$vt4$ Inst.	Total number of VM instances
$C1$: under BOTM	4	2	0	4	10
$C2$: under VM-GREEDY	6	1	1	4	12
$C3$: randomly generated	4	4	3	2	13

Table 4
Mapping schemes obtained by BOTM and VM-G (VM-GREEDY) in Experiment 1, where each cell stores (vt, DoP) for the corresponding module.

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8
BOTM	(1, 1)	(2, 1)	(2, 1)	(4, 2)	(4, 1)	(1, 1)	(4, 1)	(1, 1)	(1, 1)
VM-G	(1, 1)	(2, 1)	(4, 4)	(1, 1)	(1, 1)	(3, 1)	(1, 1)	(1, 1)	(1, 1)

w_1 filters out the headline in each data file; w_2 and w_5 calculate the average taxi in/out time at each airport, where w_2's key is the airport name and value is the taxi time, w_5's key is the airport name and value is the average taxi time; w_3 and w_6 calculate the average delay frequency of each flight, where w_3's key is the flight number and value is the delay frequency, w_6's key is the flight number and value is the average delay frequency; w_4 and w_7 calculate the frequency of each "Flight Cancellation Reason" over all of the years, where w_4's key is the cancellation code and w_4's value is 1, w_7's key is the cancellation code and w_7's value is the cancellation frequency. w_8 collects all the results in each category, where key is the ranking type (cancellation code, airport name, and flight number), and value is the result (the average taxi time, the average delay frequency, the cancellation frequency).

6.1.2. Experimental settings

We consider four VM types in Amazon Web Services (AWS) [21] and construct three different heterogeneous clusters. Table 1 tabulates the system specification and pricing model (in unit of US Dollar per minute) of each VM type, and the number of available VM instances of each VM type. In each cluster, we install STORM 1.0.0 on the VM instances, and install ZooKeeper 3.4.8 on the VM instance where Nimbus is installed. As shown in Tables 9 and 5, since the processing time of each tuple by any module of the workflow is on the order of seconds, their performance is not affected by the degradation of the virtual CPU performance, as experienced by some users running long-time jobs in AWS.

6.1.3. Performance comparison

We first execute the entire topology on one VM instance for each of four VT types in stand-alone mode to obtain the execution time matrix for one tuple on the module, as shown in Table 2. For w_1 to w_4, the time complexity of each task is $O(n)$, where n is the size of the record. For w_5 to w_8, the time complexity of each task is $O(1)$.

In the experiment, the time interval for emitting two contiguous tuples is set to be a random value within a range of [0.5 ms, 1.5 ms], and the budget is set to be five times $p(vt_4)$. We run BOTM and VM-GREEDY to obtain two mapping schemes, as tabulated in Table 4, where each cell stores (vt, DoP) for the corresponding module. For example, $(4, 2)$ for module w_3 in the mapping scheme produced by BOTM means that 2 VM instances of VM type vt_4 are used to run 2 instances of w_3.

Based on the mapping schemes produced by BOTM and VM-GREEDY, we set up two corresponding clusters $C1$ and $C2$. The $C1$ cluster produced by BOTM contains 10 VM instances, while the $C2$ cluster produced by VM-GREEDY contains 12 VM instances. We also set up a randomly generated cluster $C3$ that contains 13 VM instances satisfying the budget constraint. The configurations of these three clusters are provided in Table 3.

We run the Storm topology for flight data processing in $C1$ and $C2$ produced by BOTM and VM-GREEDY, respectively, for three times. Also, we run the topology in the default Storm system in clusters $C1$, and set the *DoP* for each module from 1 to the highest *DoP* in the mapping scheme achieved by BOTM, which is 2. Similarly, we run the topology in the default Storm system in clusters $C2$, and set the *DoP* for each module from 1 to the highest *DoP* in the mapping scheme achieved by VM-GREEDY, which is 4. In the randomly generated cluster $C3$, we set the *DoP* for each module from 1 to 4. Note that for each *DoP*, we run the experiment for three times. The performance measurements in all of these experiments are tabulated in Table 5, where the underlined throughput performance measured within a 10-minute window corresponds to the global bottleneck module. We provide such microscopic behaviors in every experiment to study the stability of each algorithm. These measurements show that the proposed BOTM algorithm achieves consistent performance in three runs while the other algorithms in comparison lack such stability.

Table 5
Throughput measurements in tuples/*min* of BOTM, VM-G (VM-GREEDY), and STORM (STORM_DEFAULT) in Experiment 1 on Flight Data, where each run lasts for 10 h.

Alg.	Idx	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	Throughput	Average Throughput
		(average number of tuples processed by each module within a 10-min window)										
BOTM	1	25 180	24 800	24 760	24 940	24 060	49 040	24 700	23 980	97 660	2398	
C1	2	24 640	24 420	24 360	24 580	23 640	48 280	24 300	23 560	95 900	2356	2389
	3	25 620	24 880	24 820	25 240	24 240	49 200	25 000	24 140	97 880	2414	
VM-G	1	19 340	19 340	19 360	19 300	18 860	38 360	19 100	18 860	76 160	1886	
C2	2	22 660	22 660	22 620	22 600	21 920	44 720	22 360	21 920	89 020	2192	1682
	3	11 460	11 460	11 440	11 440	9 680	22 620	11 320	9 680	43 580	968	
STORM	1	15 940	15 920	15 900	15 880	15 100	31 460	15 720	15 120	62 300	1510	
C1	2	16 700	16 700	16 660	16 680	15 900	32 980	16 500	15 900	65 380	1590	1573
DoP = 1	3	16 680	16 660	16 660	16 660	16 180	32 980	16 480	16 180	65 520	1618	
STORM	1	20 200	18 640	18 600	18 600	17 700	36 780	18 400	17 700	72 880	1770	
C1	2	18 140	18 140	18 140	18 120	16 860	35 860	17 940	16 860	70 640	1686	1765
DoP = 2	3	20 300	20 200	20 160	20 160	18 380	39 940	19 960	18 340	78 260	1838	
STORM	1	2 140	1 440	1 420	1 440	720	2 860	1 440	720	5 000	72	
C2	2	3 480	1 600	1 540	1 540	660	3 020	1 500	680	5 220	66	68
DoP = 1	3	2 740	1 580	1 560	1 580	660	3 120	1 560	680	5 320	66	
STORM	1	7 160	2 180	1 480	1 500	820	2 880	1 500	840	5 200	82	
C2	2	7 380	1 660	1 260	1 280	640	2 440	1 240	640	4 300	64	74
DoP = 2	3	7 540	1 480	1 460	1 460	760	2 880	1 440	740	5 020	76	
STORM	1	9 640	2 080	1 360	1 480	760	2 720	1 440	800	4 860	76	
C2	2	7 980	2 400	1 380	1 420	680	2 660	1 360	640	4 700	64	74
DoP = 3	3	9 760	2 180	1 380	1 320	820	2 700	1 300	840	4 760	82	
STORM	1	10 280	2 120	1 340	1 360	780	2 640	1 280	780	4 720	78	
C2	2	11 700	1 880	1 360	1 340	740	2 620	1 360	720	4 660	72	82
DoP = 4	3	10 960	2 200	1 360	1 300	960	2 600	2 600	2 600	2 600	96	
STORM	1	20 440	19 560	19 500	19 500	18 920	38 620	19 320	18 920	76 860	1892	
C3	2	16 700	16 720	16 660	16 660	15 860	33 040	16 520	15 860	65 420	1586	1673
DoP = 1	3	16 680	16 120	16 080	16 080	15 400	31 840	15 920	15 420	63 180	1540	
STORM	1	22 260	21 040	21 000	21 000	19 900	41 580	20 800	19 880	82 280	1988	
C3	2	19 220	17 160	17 140	17 140	16 220	33 940	16 960	16 200	67 140	1620	1320
DoP = 2	3	7 060	4 440	4 440	4 440	3 520	8 800	4 420	3 540	16 720	352	
STORM	1	5 380	5 140	5 160	5 160	2 300	10 260	5 100	2 300	17 700	230	
C3	2	5 540	4 860	4 860	4 860	2 380	9 580	4 800	2 420	16 760	238	236
DoP = 3	3	5 520	4 880	4 840	4 840	2 420	9 600	4 780	2 400	16 800	240	
STORM	1	6 520	4 160	4 100	4 120	2 180	8 200	4 100	2 160	14 420	216	
C3	2	7 120	4 960	4 960	4 960	2 320	9 800	4 920	2 340	17 040	232	228
DoP = 4	3	6 660	4 360	4 320	4 320	2 360	9 560	4 280	2 380	15 260	236	

Table 6
Execution time matrix T_e in *ms* for WRF.

	$w0$	$w1$	$w2$	$w3$	$w4$	$w5$	$w6$	$w7$
$vt1$	100.17	6889.82	3434.38	8854.28	17 591.02	55 361.93	65 107.75	1007.06
$vt2$	100.21	3592.12	1813.37	4069.33	7 821.38	29 563.50	28 720.50	586.00
$vt3$	100.28	1868.93	1710.83	2145.80	4 163.83	15 309.33	11 885.50	491.67
$vt4$	100.26	1119.35	554.91	1358.38	2 595.86	3 960.33	11 646.00	141.50

We calculate the average throughput with standard deviation across different *DoP* based on these performance measurements, and plot them in Figs. 11 and 12 for a visual comparison. We observe that BOTM consistently outperforms the other algorithms in comparison.

Both BOTM and VM-GREEDY decide the VT selection and the *DoP* for each module of the workflow. In default Storm, we vary the *DoP* for every module from 1 to the highest *DoP* among all modules in the mapping scheme produced by BOTM and VM-GREEDY. These results show that a higher *DoP* does not always yield a better workflow throughput performance as the default scheduler in Storm does not consider the global bottleneck.

6.2. Experiment 2 with climate data

6.2.1. WRF workflow

To evaluate the performance of our algorithm in real computing environments, we conduct Storm experiments based on the Weather Research and Forecasting (WRF) model [23], which has

Table 7
Storm cluster VM instances provisioned under different mapping schemes in AWS.

Cluster	$vt1$ Inst.	$vt2$ Inst.	$vt3$ Inst.	$vt4$ Inst.	Total number of VM instances
$C4$: under BOTM	4	2	0	2	8
$C5$: under VM-GREEDY	4	2	1	3	10
$C6$: randomly generated	0	2	3	3	8

Table 8
Mapping Schemes obtained by BOTM and VM-G (VM-GREEDY) in Experiment 2, where each cell stores (vt, DoP) for the corresponding module.

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7
BOTM	(1, 1)	(1, 1)	(1, 1)	(1, 1)	(2, 1)	(4, 1)	(4, 1)	(2, 1)
VM-G	(1, 1)	(2, 1)	(4, 3)	(1, 1)	(1, 1)	(3, 2)	(2, 1)	(1, 1)

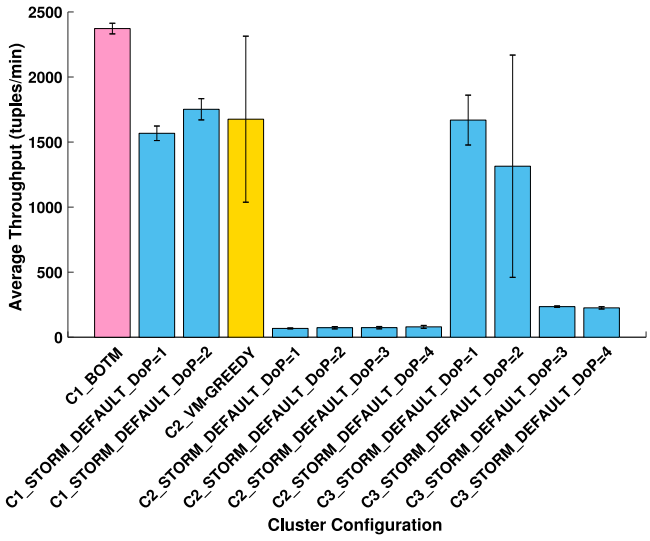


Fig. 11. The average throughput with standard deviation of the Storm topology across different degrees of parallelism (DoP) in clusters $C1$ and $C2$ produced by BOTM and VM-GREEDY, respectively, and a randomly generated $C3$ under a given budget.

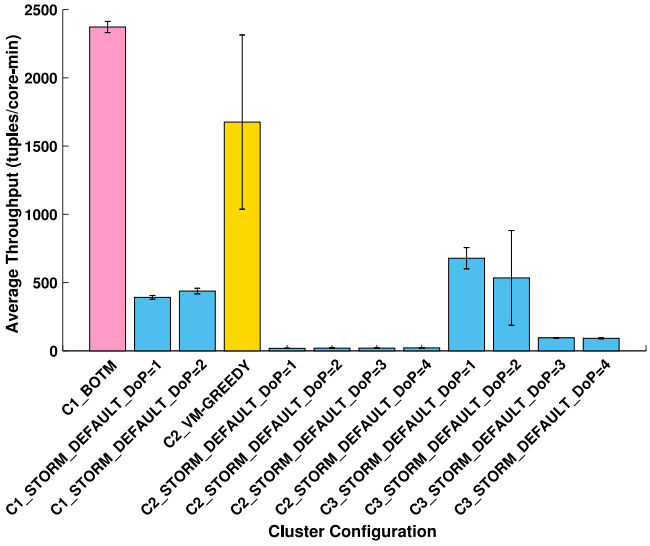


Fig. 12. The average throughput (per core) with standard deviation of the Storm topology across different degrees of parallelism (DoP) in clusters $C1$ and $C2$ produced by BOTM and VM-GREEDY, respectively, and a randomly generated $C3$ under a given budget.

been widely adopted for regional to continental scale weather forecast. The WRF model [24] generates two large classes of simulations either with an ideal initialization or utilizing real data. In our experiments, the simulations are generated from real data, which usually requires preprocessing from the WPS package [25] to provide each atmospheric and static field with fidelity appropriate to the chosen grid resolution for the model.

The structure of a general WRF workflow is illustrated in Fig. 13, where the WPS consists of three independent programs: geogrid.exe, ungrib.exe, and metgrid.exe [26]. The geogrid program defines the simulation domains and interpolates various terrestrial datasets to the model grids. The user can specify information in the namelist file of WPS to define simulation domains. that typically contain more fields than needed to initialize WRF. The ungrib program "degrib" the data and stores the results in a simple intermediate format. The metgrid program horizontally interpolates the intermediate-format meteorological data that are extracted by the ungrib program into the simulation domains defined by the geogrid program. The interpolated metgrid output can then be ingested by the WRF package, which the data by WPS, we will run the programs in WRF model. contains an initialization program real.exe for real data and a numerical integration program wrf.exe. The postprocessing model consists of ARWpost and GrADs. ARWpost reads-in WRF-ARW model data and creates output files for display by GrADS.

We duplicate three WRF pipelines each from ungrib.exe to ARWpost.exe, and group these programs into different aggregate modules to simulate real-life workflow clustering and provide various module parallelism, as shown in Figs. 14 and 15. Fig. 15 is a high-level view of grouped workflow in Fig. 14, where w_0 and w_7 are the start and end modules [26].

We execute the WRF topology in the same computing environment as the experiments for flight data processing.

6.2.2. Performance comparison

We first execute the entire topology on one VM instance of each of four VT types in the stand-alone mode to obtain the execution time matrix for one tuple on the module, as shown in Table 6.

Similarly, in this set of experiments, the time interval for emitting two contiguous tuples is set to be a random value within a range of [0.5 ms, 1.5 ms]. The budget is set to be five times $p(vt_4)$. We run BOTM and VM-GREEDY to obtain two mapping schemes as tabulated in Table 8. Similar to Table 4, each cell stores (vt, DoP) for each corresponding module. Based on the mapping schemes produced by BOTM and VM-GREEDY, we set up two corresponding clusters $C4$ and $C5$. The $C4$ cluster produced by BOTM contains 8 VM instances, while the $C5$ cluster produced by VM-GREEDY contains 10 VM instances. We also set up a randomly generated cluster $C6$ that contains 8 VM instances satisfying the budget constraint. The configurations of these three clusters are provided in Table 7.

We run the Storm topology for WRF workflow in $C4$ and $C5$ produced by BOTM and VM-GREEDY, respectively, for three times. Also, we run the topology in the default Storm system in clusters $C4$, and set the *DoP* for each module to be 1, which is the highest *DoP* in the mapping scheme achieved by BOTM. Similarly, we run the topology in the default Storm system in clusters $C5$, and set the *DoP* for each module from 1 to the highest *DoP* in the mapping scheme achieved by VM-GREEDY, which is 3. In the randomly generated cluster $C6$, we set the *DoP* for each module from 1 to 3. For each *DoP*, we run the experiment for three times. All performance measurements are tabulated in Table 9, where the underlined throughput performance measured within a 10-min window corresponds to the global bottleneck module.

We calculate the average throughput with standard deviation across different degrees of parallelism based on these performance measurements, and plot them in Figs. 16 and 17 for a visual comparison. Again, we observe that BOTM consistently outperforms the other algorithms in comparison. In Fig. 18, we also illustrate the resource consumption (number of cores $\times$ memory size $\times$ time unit) for WRF data processing across different degrees of parallelism (DoP) in clusters C4, C5, and randomly gen-

Table 9
Throughput measurements in tuples/*hour* of BOTM, VM-G (VM-GREEDY), and STORM (STORM_DEFAULT) in Experiment 2 on WRF workflow, where each run lasts for 10 h.

Algorithm	Idx	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	Throughput	Average Throughput
		(average number of tuples processed by each module within a 10-min window)									
BOTM	1	960	440	900	400	200	60	40	100	240	
C4	2	940	460	920	400	200	40	40	100	240	240
	3	940	440	900	380	220	40	60	100	240	
VM-G	1	875	482	71	393	196	54	36	120	216	
C5	2	893	429	321	339	339	36	36	89	216	210
	3	810	479	397	380	380	33	50	99	198	
STORM	1	480	21	42	10	10	10	10	10	60	
C4	2	920	60	140	60	60	20	20	40	120	66
DoP = 1	3	137	9	24	9	23	3	3	3	18	
STORM	1	531	11	23	34	34	11	11	22	66	
C5	2	133	9	17	9	9	3	3	3	18	36
DoP = 1	3	164	7	14	7	7	4	4	4	24	
STORM	1	261	20	38	14	6	3	6	6	18	
C5	2	209	5	9	5	9	2	5	5	12	22
DoP = 2	3	564	58	122	58	38	13	6	13	36	
STORM	1	485	4	11	4	20	4	4	8	24	
C5	2	631	28	42	19	14	14	14	5	30	24
DoP = 3	3	380	20	12	12	3	3	6	3	18	
STORM	1	375	17	25	8	17	8	8	8	48	
C6	2	143	3	3	3	6	3	3	6	18	28
DoP = 1	3	120	5	8	3	10	3	3	3	18	
STORM	1	297	13	30	13	13	3	7	7	18	
C6	2	281	24	46	17	12	4	3	6	18	18
DoP = 2	3	245	3	11	5	11	3	5	5	18	
STORM	1	812	18	36	18	61	18	24	48	108	
C6	2	137	2	4	1	5	3	3	7	6	48
DoP = 3	3	345	8	16	10	23	8	5	13	30	

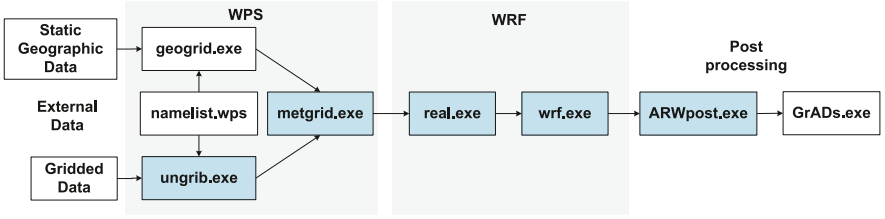


Fig. 13. A general structure of the executable WRF workflow.

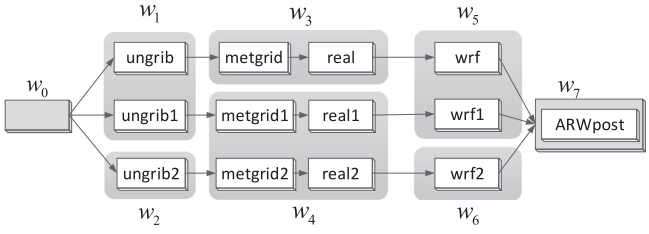


Fig. 14. The WRF Storm workflow of three pipelines in the experiments.

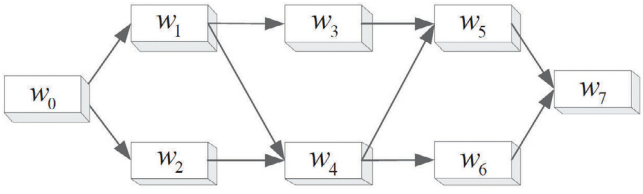


Fig. 15. The WRF Storm workflow after grouping.

erated cluster C6 by BOTM, VM-GREEDY and STORM_DEFAULT, respectively, under a given budget.

In this experiment, we observe that the *DoP* for each module in the mapping scheme produced by BOTM is only 1. However, we still run Storm in its default setting in the cluster provided by BOTM and increase the *DoP* from 1 to the highest degree decided by VM-GREEDY. These results show that even without parallel processing, BOTM still outperforms the other algorithms with parallel processing.

6.3. Summary

The performance superiority of BOTM is brought by a careful design that follows two important guidelines: (i) it is bottleneck-oriented as the global bottleneck module determines the overall throughput of the entire workflow, and (ii) it is bottleneck-adaptive as the global bottleneck may shift to a different module after each adjustment and the most suitable adjustment is adopted to maximize the global throughput of the workflow instead of the local throughput of any component module. Storm's

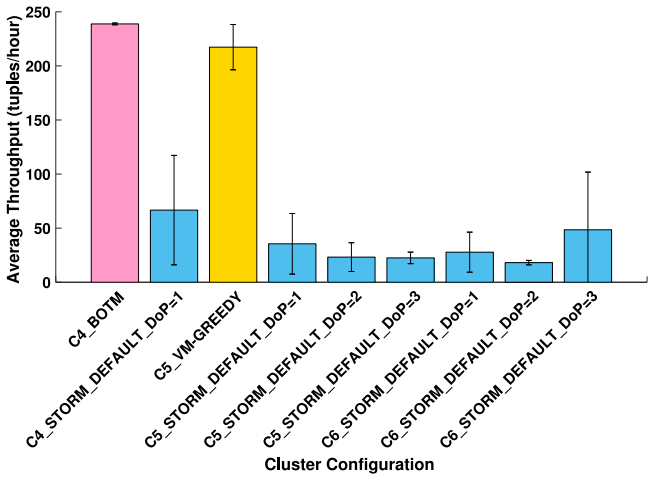


Fig. 16. The average throughput with standard deviation of the Storm topology for WRF data processing across different degrees of parallelism (DoP) in clusters C4 and C5 produced by BOTM and VM-GREEDY, respectively, and randomly generated cluster C6 under a given budget.

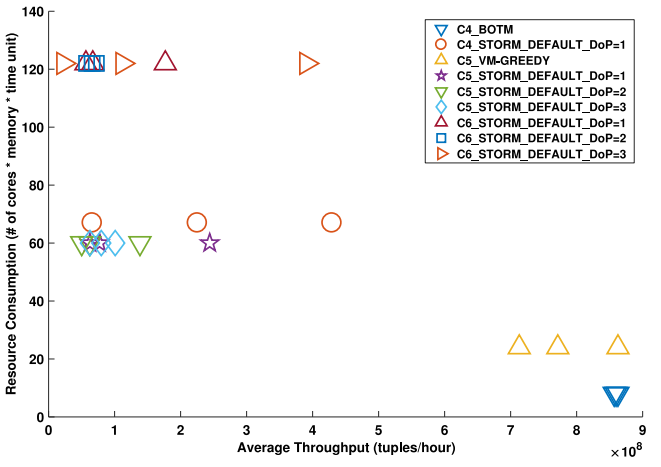


Fig. 18. The resource consumption for WRF data processing across different degrees of parallelism (DoP) in clusters C4, C5, and randomly generated cluster C6 by BOTM, VM-GREEDY and STORM_DEFAULT, respectively, under a given budget.

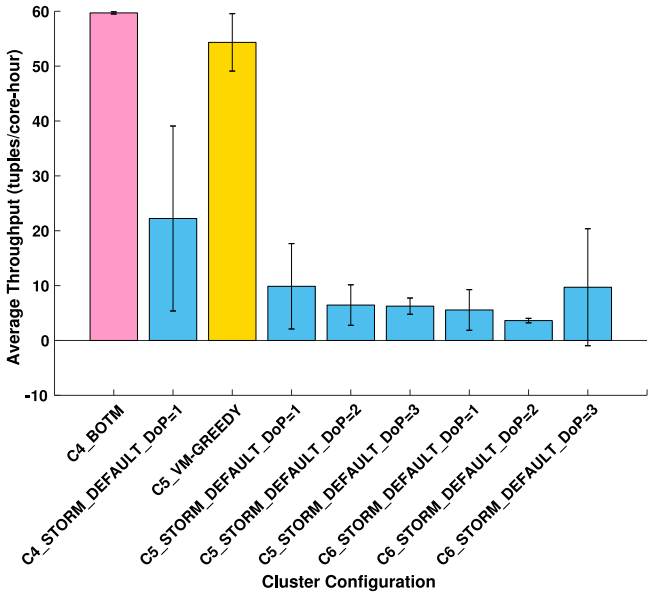


Fig. 17. The average throughput (per core) with standard deviation of the Storm topology for WRF data processing across different degrees of parallelism (DoP) in clusters C4 and C5 produced by BOTM and VM-GREEDY, respectively, and randomly generated cluster C6 under a given budget.

default scheduler (STORM_DEFAULT) neither considers the bottleneck module nor performs selective resource allocation; VM-GREEDY also neglects the bottleneck and only allocates resources to modules in a topologically sorted order.

7. Conclusion

We formulated a budget-constrained Storm topology mapping problem to maximize the throughput in cloud environments, referred to as STM-BC, which was shown to be NP-complete. We designed a heuristic algorithm BOTM for STM-BC and demonstrated its performance superiority over other methods through extensive simulations and experiments.

Both of the experiments were conducted on real-life datasets, but with some differences. Firstly, different data volumes were tested for scalability evaluation. The flight statistics workflow processes 22 years of global flight datasets of about 12 GB, and the WRF workflow processes less than 1 GB data. This difference in scale was also reflected by their throughput measurements: the throughput of the flight statistics workflow is around thousands of tuples per minute, and the throughput of the WRF workflow is around hundreds of tuples per hour. Secondly, different topologies, which imply different densities and processing dynamics, were tested to illustrate the robustness of our proposed algorithm.

It would be of our future interest to refine and generalize the mathematical models to achieve a higher level of accuracy for workflow execution time measurement in real-world clouds. Moreover, in real networks, physical servers may fail under a certain probability and the actual workload of workflow modules may be subject to dynamic changes, which will be considered in our future work.

CRediT authorship contribution statement

Huiyan Cao: Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing - original draft, Writing - review & editing. **Chase Q. Wu:** Conceptualization, Funding acquisition, Formal analysis, Investigation, Project administration, Methodology, Resources, Supervision, Writing - original draft, Writing - review & editing. **Liang Bao:** Data curation, Software, Formal analysis, Investigation, Methodology. **Aiqin Hou:** Software, Validation, Methodology. **Wei Shen:** Formal analysis, Validation, Methodology.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research is sponsored by U.S. National Science Foundation under Grant No. CNS-1828123 with New Jersey Institute of Technology.

References

[1] Apache, Storm, 2016, http://storm.apache.org.
[2] Flink. https://flink.apache.org/.
[3] Spark Streaming. https://spark.apache.org/streaming/.
[4] Samza. http://samza.apache.org/.
[5] Apex. https://apex.apache.org/.
[6] Google Cloud Dataflow. https://cloud.google.com/dataflow/.
[7] L. Neumeyer, B. Robbins, A. Nair, A. Kesari, Yahoo s4: Distributed stream computing platform, in: Data Mining Workshops (ICDMW), 2010 IEEE International Conference on, 2010, pp. 170–177.
[8] Jstorm. http://jstorm.io/.
[9] Heron. https://twitter.github.io/heron/.
[10] B. Agarwalla, N. Ahmed, D. Hilley, U. Ramachandran, Streamline: a scheduling heuristic for streaming application on the grid, in: Proc. of the 13th Multimedia Comp. and Net. Conf., San Jose, CA, 2006.
[11] A. Sekhar, B. Manoj, C. Murthy, A state-space search approach for optimizing reliability and cost of execution in distributed sensor networks, in: Proc. of Int. Workshop on Dist. Comp., 2005 pp. 63–74.
[12] B. Peng, M. Hosseini, Z. Hong, R. Farivar, R. Campbell, R-storm: Resource-aware scheduling in storm, in: Proceedings of the 16th Annual Middleware Conference, ACM, 2015, pp. 149–161.
[13] L. Eskandari, Z. Huang, D. Eyers, P-scheduler: Adaptive hierarchical scheduling in apache storm, in: Proceedings of the Australasian Computer Science Week Multiconference, ACM, 2016, p. 26.
[14] Z. Chen, J. Xu, J. Tang, K. Kwiat, C. Kamhoua, G-storm: GPU-enabled high-throughput online data processing in storm, in: Big Data (Big Data), 2015 IEEE International Conference on, IEEE, 2015, pp. 307–312.
[15] J. Xu, Z. Chen, J. Tang, S. Su, T-storm: Traffic-aware online scheduling in storm, in: Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on, 2014, pp. 535–544.
[16] L. Aniello, R. Baldoni, L. Querzoni, Adaptive online scheduling in storm, in: Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems, 2013, pp. 207–218.
[17] V. Cardellini, V. Grassi, F. Lo Presti, M. Nardelli, Distributed qos-aware scheduling in storm, in: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, ACM, 2015, pp. 344–347.
[18] Q. Wu, Y. Gu, Optimizing end-to-end performance of data-intensive computing pipelines in heterogeneous network environments, J. Parallel Distrib. Comput. 71 (2) (2011) 254–265.
[19] Bottleneck-Oriented Topology Mapping (BOTM). https://github.com/BigDataCenter/BOTM.
[20] C. Wu, H. Cao, Optimizing the performance of big data workflows in multi-cloud environments under budget constraint, in: Proceedings of the 13th IEEE International Conference on Services Computing, San Francisco, USA, 2016.
[21] Amazon, EC2, 2016, http://aws.amazon.com/ec2/.
[22] ASA Sections on:Statistical Computing Statistical Graphics, http://stat-computing.org/dataexpo/2009/the-data.html.
[23] W. Skamarock, J. Klemp, J. Dudhia, D. Gill, D. Barker, M. Duda, X. Huang, W. Wang, J. Powers, A Description of the Advanced Research WRF Version 3, Tech. Rep. NCAR/TN-475+STR, National Center for Atmospheric Research, Boulder, Colorado, USA, 2008.
[24] Weather research and forecasting (wrf) model, 2020, http://wrf-model.org/index.php.
[25] Wrf preprocessing system (wps), 2020, http://www2.mmm.ucar.edu/wrf/users/wpsv2/wps.html.
[26] C. Wu, X. Lin, D. Yu, W. Xu, L. Li, End-to-end delay minimization for scientific workflows in clouds under budget constraint, IEEE Trans. Cloud Comput. 3 (2) (2015) 169–181.

Huiyan Cao received the B.S. degree in software engineering from XiDian University, P.R. China and the M.S. degree in Information, Production, and Systems Engineering from Waseda University, Japan, in 2014. She is currently a doctoral student in the Department of Computer Science at New Jersey Institute of Technology, and works in the Big Data Group. Her research interests include big data, cloud computing, and computer networks.

Chase Q. Wu completed his Ph.D. dissertation at Oak Ridge National Laboratory and received his Ph.D. degree in computer science from Louisiana State University in 2003. He was a research fellow at Oak Ridge National Laboratory during 2003–2006 and an associate professor at University of Memphis during 2006–2015. He is currently an associate professor at New Jersey Institute of Technology. His research interests include big data, parallel and distributed computing, high-performance networking, sensor networks, and cyber security.

Liang Bao received the PhD degree in computer science from Xidian University, P.R. China, in 2010. He is currently an associate professor with the School of Computer Science and Technology, XiDian University. His research interests include software architecture, cloud computing and big data. He is a member of the IEEE.

Aiqin Hou received the Ph.D. degree in the School of Information Science and Technology at Northwest University, China, in 2018. She is currently an associate professor with the School of Information Science and Technology, Northwest University, China. Her research interests include big data, high-performance network, and bandwidth scheduling.

Wei Shen received the Ph.D. degree from College of Computer Science and Technology, Zhejiang University, China. He is currently a professor with the Department of Computer Science at Zhejiang Sci-Tech University, China. His research interests include computation theory, artificial intelligence, and big data.