Performance-aware deployment of streaming applications in distributed stream computing systems

Dawei Sun*

School of Information Engineering,
China University of Geosciences,
Beijing, 100083, China
and
Polytechnic Center for Territory Spatial Big-data,
MNR of China, China
Email: sundaweicn@cugb.edu.cn
*Corresponding author

Shang Gao

School of Information Technology,
Deakin University,
Victoria 3216, Australia
Email: shang.gao@deakin.edu.au

Xunyun Liu

Cloud Computing and Distributed Systems (CLOUDS) Laboratory,
School of Computing and Information Systems,
The University of Melbourne, Australia
Email: xunyunliu@gmail.com

Fengyun Li

School of Computer Science and Engineering,
Northeastern University,
Shenyang, 110819, China
Email: lifengyun@mail.neu.edu.cn

Rajkumar Buyya

Cloud Computing and Distributed Systems (CLOUDS) Laboratory,
School of Computing and Information Systems,
The University of Melbourne, Australia
Email: rbuyya@unimelb.edu.au

Abstract: Performance-aware deployment of streaming applications is one of the key challenging problems in distributed stream computing systems. We proposed a performance-aware deployment framework (Pa-Stream) for distributed stream computing systems. By addressing the important aspects of the framework, this paper makes the following contributions: 1) investigated the performance-aware deployment of a streaming application over distributed and heterogeneous computing nodes, and provided a general application deployment model; 2) demonstrated a streaming applications deployment scheme by proposing an artificial bee colony strategy that deploys application's vertices onto the best set of computing nodes; an incremental online redeployment strategy was used to redeploy the running application; 3) developed and integrated Pa-Stream into Apache Storm platform; 4) evaluated the fulfilment of low latency and high throughput objectives in a distributed stream computing environment. Experimental results demonstrate that the proposed Pa-Stream provided effective performance improvements on latency, throughput and resource utilisation.

Keywords: performance awareness; application deployment; stream computing; artificial bee colony algorithm; distributed system.

Reference to this paper should be made as follows: Sun, D., Gao, S., Liu, X., Li, F. and Buyya, R. (2020) 'Performance-aware deployment of streaming applications in distributed stream computing systems', *Int. J. Bio-Inspired Computation*, Vol. 15, No. 1, pp.52–62.

Biographical notes: Dawei Sun is an Associate Professor at the School of Information Engineering, China University of Geosciences, Beijing, China. His current researches interests include big data computing, cloud computing and distributed systems.

Shang Gao is currently a Senior Lecturer at the School of Information Technology, Deakin University, Geelong, Australia. Her current research interests include distributed collaboration, cybersecurity and cloud computing.

Xunyun Liu received his BE and ME in Computer Science and Technology from the National University of Defense Technology in 2011 and 2013, respectively. He obtained his PhD in Computer Science at the University of Melbourne in 2018. His research interests include stream processing and distributed systems.

Fengyun Li is an Associate Professor at the School of Computer Science and Engineering, Northeastern University, China. Her current researches interests include distributed systems, network security, privacy preserving and cloud computing.

Rajkumar Buyya is a Redmond Barry Distinguished Professor and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at University of Melbourne, Australia. He is also serving as the Founding CEO of the Manjrasoft, a spin-off company of the university, commercialising its innovations in Cloud computing. He has authored over 650 publications and four text books. He is one of the highly cited authors in computer science and software engineering worldwide (h-index 128, 85,600+ citations). He has served as the Founding Editor-in-Chief of the *IEEE Transactions on Cloud Computing* and now serving as EiC of *Journal of Software: Practice and Experience.*

1 Introduction

In the big data era, stream computing is an on-the-fly computing paradigm, which can timely extract valuable information from input data stream. As data streams are directly processed without storage in advance, stream computing is a good choice for event-driven applications that require real-time reaction to events in the form of fluctuating data streams. It is gaining tremendous attention due to an increasing number of emerging application scenarios (Eidenbenz and Locher, 2016) that rely heavily on live and real-time processing of data streams. In the field of finance, stream computing can be used to detect specific events in high-frequency trading in a timely manner. In the field of social networks, stream computing can be used to collect current hot topics in real-time, and promote them to the online users. In the field of system monitoring, stream computing can be used to monitor data access online, and issue real-time warning of abnormal access. In the field of smart cities, stream computing can be used for intelligent transportation. Stream computing can also be used for real-time product recommendations during online shopping.

To address the specific issues raised by stream data featured with high volume and high velocity, a new generation of stream computing system has been developed. Storm, as one of the most popular open source systems (Li et al., 2016, 2017). It keeps the system average latency within the range of milliseconds level.

The new generation of stream computing systems usually offer simple programming interfaces for users to design their streaming applications. In Storm, the programming user interfaces are called spout and bolt, and the streaming application logic is abstracted as a streaming topology. The topology is a directed acyclic graph (DAG).

Vertices in a DAG represent application operators (encapsulating logics such as, filtering, aggregation or merging) as well as specific computations. Each vertex processes one or more input tuples from the upstream vertices according to its implemented function, and creates one or more output tuples in the form of data stream for downstream vertices. Edges in a DAG indicate communications channels between those vertices. We treat the terms streaming application, streaming topology, and DAG as synonymous. A DAG is submitted to a stream computing system and deployed on one or many computing nodes. Stream applications can be either dependent or independent according to the relationships among them; stream applications can also be either preemptive or non-preemptive depending on whether they can be interrupted during execution. Computing nodes can be either homogeneous or heterogeneous hosts depending on whether the computing nodes have the same properties or not.

Low latency is one of the critical performance requirements for a stream computing system. To achieve it, usually application deployment is one of the key processes to start with. Users are often required to deploy inter-dependent operators to a set of available computing nodes to satisfy objective constraints, such as some certain throughput and latency. The problem of application deployment in distributed big data stream computing

systems is also one of the most thought-provoking NP-hard problems in general cases (Cai et al., 2019b; Cui et al., 2019b; Kotto-Kombi et al., 2017). The volume of data streams fluctuates over time in an unpredictable manner, and the amount of available resources is also dynamically changing over time. Stream computing systems are expected to adapt to the dynamic heterogeneous resources to meet resource needs of streaming applications. It is disadvantageous to over or under-provision resources over a long period of time. A performance-aware stream computing system should be timely aware of the changes of each computing node and the volume of the multiple data streams. All of these requirements make the optimal application deployment challenging.

To utilise resources wisely, a fundamental solution is performance-aware deployment that allows the application to scale out or scale in according to the processing needs brought by fluctuating data streams. However, the majority of state-of-the-art solutions (Hirzel et al., 2014; Thoman et al., 2018) do not provide an efficient performance-aware deployment strategy that knows how to deploy operators to available resources for the current data stream. Previous work in this area mainly focuses on scheduling. E.g., Storm only supports user-defined parallelism for each vertex, and deploys a DAG on a fixed number of computing nodes with simple polices, such as the default round-robin strategy, which lack performance awareness and adaptation capabilities.

A performance-aware deployment strategy should be able to decide when and how streaming applications are to be deployed with regard to the specific applications structures and available resources. To achieve this goal, a performance-aware deployment strategy needs to know the available resources of each computing node, the dependencies between operators, and how to deploy inter-dependent operators to the best set of available computing nodes. Currently, the requirements of performance-aware deployment are not fully considered (Dias de Assunção et al., 2017). This creates the demand for investigating a performance-aware deployment framework for running streaming applications in distributed stream computing systems, processing data streams in an efficient manner, minimising the system latency and maximising the system throughputs.

1.1 Key contributions

This paper makes the following contributions:

1 Investigated performance-aware deployment of a streaming application over distributed and heterogeneous computing nodes, and provided a system model and application deployment model (Pa-Stream).

2 Demonstrated the proposed deployment scheme (Pa-Stream) based on artificial bee colony (ABC) algorithm to deploy vertices of a streaming application onto the best set of computing nodes in an acceptable amount of time; and redeployed the runtime application with an incremental online redeployment strategy to improve latency and throughput.

3 Developed and integrated Pa-Stream into Apache Storm platform.

4 Evaluated the fulfilment of lower latency, higher throughput and resource utilisation objectives with two streaming applications Top_N and WordCount.

1.2 Paper organisation

The rest of the paper is organised as follows: Section 2 introduces the four steps of application deployment in a stream computing system. Section 3 describes the system model of Pa-Stream. Section 4 formalises the latency modelling of stream application and the stream application scheduling model. Section 5 focuses on the system architecture, the modified ABC algorithm, and the incremental online redeployment algorithm in Pa-Stream. Section 6 introduces the experimental environment and performance evaluation of Pa-Stream. Section 7 reviews the related work on performance-aware deployment in distributed computing systems and ABC-based applications. Finally, conclusions and future work are given in Section 8.

2 Background

Storm, as one of the most popular distributed stream computing systems, provides millisecond-level response to users. The deployment of stream applications is a key to achieving low latency. Usually, a stream application is deployed on Storm platform with the following steps.

2.1 DAG creation

Functions of a stream application are usually divided into a set of data-dependent operators, described as a DAG, and termed a topology in Storm. Each vertex in the DAG has a special function to process one or more input data streams from upstream vertices, and output the processed data streams to downstream vertices. The group mode of data stream is a strategy used to decide how the data tuples are distributed from upstream vertices to downstream vertices, and can be set by data stream grouping interface for each vertex. Some group modes have been provided by Storm, such as fieldsGrouping and globalGrouping.

All the vertices are implemented through two interfaces in Storm, spout and bolt. Spout interface is used to implement a vertex that receives data from the external data source. Bolt interface is used to implement a vertex that processes data streams. Its corresponding processing logic is defined by users with regard to its functional requirements.

2.2 Parameter configuration

There are a series of stream application parameters to be set. For example, the parallelism degree of a vertex, which determines the number of instances of a vertex, and plays a

key role to implementing elastic computing. However, it is extremely hard to determine a proper parallelism degree for a vertex in a real-time computing environment, as there are many factors such as input rates of data streams, time complexity of each vertex, and available resources in a data centre.

Current practice on Storm platform is to set most of the parameters based on user's experience and it is not allowed to readjust or re-optimise the topology while the application is running. However, the configuration of the parameters is not one size fits all. It needs to be readjusted and re-optimised in response to the changes of the computing environment. The problem is beyond the scope of this paper but relevant studies can be found in Mencagli et al. (2018) and Wang et al. (2017).

2.3 Stream application deployment

Stream applications need to be deployed onto the available computing nodes in a data centre. Each vertex of a DAG is assigned to an appropriate computing node. In Storm platform, a deployment system consists of a nimbus node, one or more supervisors nodes, and one or more zookeepers nodes. Nimbus node is used to receive and manage stream applications. Supervisor node is used to run and manage instances of vertex on worker processes. Zookeeper is used to coordinate states between supervisor and nimbus.

Application deployment is a process of deploying inter-dependent sub-tasks of an application onto a set of available computing nodes while ensuring the execution satisfies user's specified service level agreement (SLA) constraints. There is an increasing research interests on this SLA-fulfilling problem (Dias de Assunção et al., 2017; Thoman et al., 2018), but it has not been completely solved and still requires further investigation.

2.4 Online running and optimisation

A data stream in distributed stream computing environments has two typical features (Mencagli et al., 2018): the first one is non-stationarity, which states that the volume of stream varies constantly due to the fact that data stream arrives at different rates, and often measured in seconds; the other feature is burstiness, as data stream may happen at multiple time scales and is measured by second-order properties, such as the index of dispersion or the Hurst parameter.

A stream application receives continuous data streams and processes them in an online fashion. To keep the efficiency and adaptability of an elastic stream computing system, an online running and optimisation strategy is required to handle data stream fluctuation and changes in available resource.

In this paper, we focus on performance-aware deployment of streaming applications in Strom platform. A heuristics deployment strategy is employed in the process of deployment and online optimisation. A sub-optimal scheduling scheme can be obtained through the proposed deployment strategy, allowing application to scale out or scale in according to the fluctuating data stream and changing resources.

3 System model

In this section, system model are described, including single stream application model, and multiple stream applications model.

3.1 Single stream application model

The topology of a stream application (Lombardi et al., 2018) can be described by a DAG $G = (V(G), E(G))$, where $V(G) = \{v_i \mid i \in 1, \cdots, n\}$ is a finite set of n vertices, $E(G) = \{e_{v_i,v_j} \mid v_i, v_j \in V(G)\}$ is a finite set of directed edges.

Vertex $v_i \in V(G)$ is a computing task of the stream application, which receives input data streams, processes data tuples, and then generates new output data streams. One or more instances of v_i can be created on-demand, that is $v_i = \{v_{i1}, v_{i2}, \cdots, v_{im}\}$, $m \in \{1, 2, \cdots\}$. For each vertex instance v_{ij}, it is an atomic computing task, and can only be executed on one computing node. The weight w_{v_i} associated with vertex v_i is the computation cost of v_i, which indicates the cost that v_i takes to process input data tuples. Weights of all m instances of vertex v_i are the same, that is $w_{v_i} = w_{v_{i1}} = w_{v_{i2}} = \cdots = w_{v_{im}}$.

An edge $e_{v_i,v_j} \in E(G)$ is a directed edge from v_i to v_j, where v_i and v_j are directly connected vertices, v_i is the upstream vertex of v_j, and v_j is the downstream vertex of v_i. For instance, an edge $e_{v_{i1},v_{j2}} \in E(G)$ is a directed edge connecting v_{j1} to v_{j2}. The weight $w_{e_{v_i,j}}$ associated with edge e_{v_i,v_j} is the communication cost of e_{v_i,v_j}, , which indicates the cost that e_{v_i,v_j} takes to transfer output data tuple of v_i from v_i to v_j. Especially, if v_i and v_j are running on the same computing node, the time of transferring output data tuple of v_i from v_i to v_j is usually considered to be 0. Weights of all m instances of vertex v_i and all n instances of vertex v_j are the same, which is formulated as the following: $w_{e_{vi1,j2}} = w_{e_{vi2,j1}} = w_{e_{vi3,j5}} = \cdots = w_{e_{vim,jn-2}}$.

A vertex $v_{in,G}$ without a directed edge from any other vertices in $V(G)$ to $v_{in,G}$ is named input vertex of G, and a vertex $v_{out,G}$ without a directed edge from $v_{out,G}$ to other any vertices in $V(G)$ is named output vertex of G. For the sake of simplicity and without loss of generality, we assume that there is only one input vertex and only one output vertex in each stream application.

3.2 Multiple stream applications model

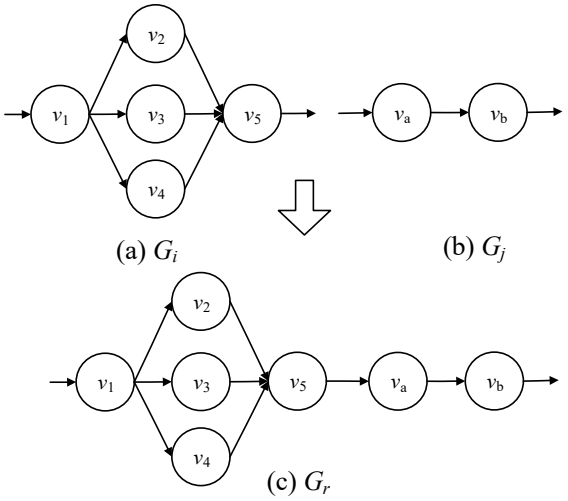
There could be multiple stream applications. To simplify the deployment of multiple stream applications, we try to combine multiple stream applications into a single stream application. The relationship between any two stream

applications can be subdivided into two categories: data dependent and non-data dependent.

In a data dependent scenario, the output of a steam application G_i is the input of the other stream application G_j. The result of the combined single stream application G_r can be obtained by combining G_i and G_j according to the data dependencies. That is, $V(G_r) = V(G_i) \cup V(G_j)$, $E(G_r) = E(G_i) \cup E(G_j) \cup e_{v_{out,G_i}, v_{in,G_j}}$, where $v_{out,G_i} \in G_i$, is the output vertex of G_i; $v_{in,G_j} \in G_j$, is the input vertex of G_j; and $e_{v_{out,G_i}, v_{in,G_j}}$ is a new directed edge from v_{out,G_i} to v_{in,G_j}. The input vertex v_{in,G_i} of G_i is the input vertex of G_r, and the output vertex v_{out,G_j} of G_j is the output vertex of G_r.

As shown in Figure 1, steam application G_i and G_j are combined into a new result steam application G_r. The output of vertex of v_5 is the input of vertex of v_a in the result steam application G_r. A new directed edge from v_5 to v_a is created in G_r.

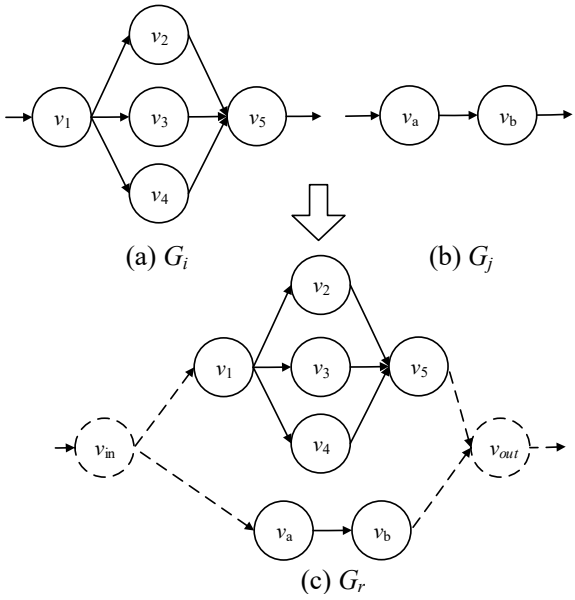
Figure 1 Combing multiple stream applications with data dependency into one single stream application



In a non-data dependency scenario, the resulting single stream application G_r can be obtained by adding new input vertex v_{in,G_r}, output vertex v_{out,G_r} and corresponding new directed edges for G_i and G_j. That is, $V(G_r) = V(G_i) \cup V(G_j) \cup v_{in,G_r} \cup v_{out,G_r}$, the weight $w_{v_{in,G_r}}$ of input vertex v_{in,G_r} is 0, that is $w_{v_{in,G_r}} = 0$. Similarly, the weight $w_{v_{out,Gr}}$ of output vertex v_{out,G_r} is also 0, that is $w_{v_{out,G_r}} = 0$. $E(G_i) \cup E(G_j) \cup e_{v_{in,G_r}, v_{in,G_i}} \cup e_{v_{in,G_r}, v_{in,Gj}} \cup e_{v_{out,Gi}, v_{out,G_r}} \cup e_{v_{out,Gj}, v_{out,G_r}}$. $e_{v_{in,G_r}, v_{in,G_i}}$, $e_{v_{in,G_r}, v_{in,Gj}}$, $e_{v_{out,Gi}, v_{out,G_r}}$ and $e_{v_{out,Gj}, v_{out,G_r}}$ are new directed edges from v_{in,G_r} to v_{in,G_i}, from v_{in,G_r} to v_{in,G_j}, from v_{out,G_i} to v_{out,G_r}, and from v_{out,G_j} to v_{out,G_r}, respectively. The weight of new directed edges are also 0. The input of G_r is the combination of the original input for G_i and G_j, the output of G_r is the combination of the original input for G_i and G_j.

As shown in Figure 2, steam application G_i and G_j are now merged into a new result steam application G_r. A new input vertex v_{in} and output vertex v_{out} are created for G_r. There are four new directed edges added, which are from v_{in} to v_1, from v_{in} to v_a, from v_5 to v_{out}, and from v_b to v_{out} are created for G_r. Weight of those new vertices and edges are 0.

Figure 2 Combing multiple stream applications without data dependency into single stream application



For applications with both data dependency and non-data dependency relationships, a complex result stream application can be obtained by applying the above two combination rules consecutively.

4 Problem formulation

In this section, we formularies latency of stream application, stream application scheduling model, and utility quantification of constraints in distributed stream computing systems.

4.1 Latency of stream application

As the topology of a stream application is described by a DAG G, all vertices in G are topologically ordered, forming one or more directed paths.

A directed path from v_i to v_j can be describe as $p_{v_i,v_j} = < v_i, \cdots, v_j >$, the latency of p_{v_i,v_j} is the total elapsed time calculated by adding up the computing time of each vertex and the communication time of each edge on the path p_{v_i,v_j}. It can be formulated by equation (1).

$$l\left(p_{v_i,v_j}\right) = \sum_{v_k \in p_{v_i,v_j}} w_{v_k} + \sum_{e_{v_i,v_k} \in p_{v_i,v_j}} w_{e_{v_i,v_k}}. \quad (1)$$

The latency of G is the maximum latency of all directed paths $P(G)$ of G. It can be described as equation (2).

$$l(G) = \max_{p \in P(G)} \left(l(p)\right). \quad (2)$$

The maximum latency of a directed path starting from input vertex v_{in} and ending at output vertex v_{out} of G, can be further described as equation (3).

$$l(G) = \max_{p_{v_{in},v_{out}} \in P_{v_{in},v_{out}}(G)} \left(l\left(p_{v_{in},v_{out}} \right) \right) \tag{3}$$

where $P_{v_{in},v_{out}}(G)$ is the directed path set of G, with all its paths starting at the input vertex v_{in} and ending at the output vertex v_{out}.

The latency $l(G)$ of a stream application G is one of the key measurements to evaluate the performance of a distributed stream computing system.

4.2 Stream application scheduling model

In a stream computing system, application scheduling is a dynamic problem to be solved at runtime (Cardellini et al., 2016), which is subject to periodic or unpredictable data stream fluctuations and the changing availability of resources. This long-run problem further exaggerates the difficulty of application scheduling.

For a stream application G and a set of computing nodes CN, the objective of scheduling vertices of stream application G onto interconnected computing nodes CN in data centre is to find a scheduling s: $V(G) \rightarrow CN$ that minimises $l(G)$, satisfying user specified SLA constraints such as low system latency and high system throughput.

Let $DC = \{cn_1, cn_2, \cdots, cn_{n_{cn}}\}$ be a set of computing nodes in a data centre, and $U = \{u_1, u_2, \cdots, u_{m_u}\}$ be a set of users. For the sake of simplicity, and without loss of generality, we assume that each user submits only one application, the stream application submitted by the i^{th} user u_i is denoted as G_i. The DAG scheduling problem of all m_u DAGs in the data centre is then formulated as follows:

$$\min \sum_{i=1}^{m_u} l(G_i) \tag{4}$$

subject to

$$l(G_i) \le l_{\max}(G_i), \forall u_i \in U \tag{5}$$

$$t(G_i) \ge t_{\min}(G_i), \forall u_i \in U \tag{6}$$

in which $l(G_i)$ and $t(G_i)$ are the latency and throughput of the i^{th} application in the user set, respectively. $l_{\max}(G_i)$ and $t_{\min}(G_i)$ are the maximum latency and minimum throughput of the i^{th} application, respectively, which are user-specified SLAs constraints. Specifically, throughput of an application is the amount of data tuple processing per unit time.

5 PA-Stream: architecture and algorithms

The above analysis builds up the foundation of Pa-Stream – a performance-aware deployment framework. In this section, we discuss the overall structure of Pa-Stream, including its system architecture, modified ABC algorithm, and incremental online redeployment algorithm.

5.1 System architecture

The system architecture of Pa-Stream is composed of one nimbus, some zookeepers, and a bunch of supervisors.

Nimbus receives the streaming applications from the users, creates the instance graph according to the logic graph and configuration parameters, and then deploys vertex in-stances to workers on appropriate supervisors following the specific deployment strategy. Different deployment strategies can be employed via IScheduler interface. Storm 1.1.0 or above (Toshniwal et al., 2014) supports four kinds of built-in deployment strategies: DefaultScheduler, Isolation-Scheduler, Mul-titenantScheduler, and ResourceA-wareScheduler. Configurations in Storm.yaml specify the selected deployment strategy. We implement Pa-Stream using the IScheduler interface. Pa-Stream is deployed on nimbus and in charge of the initial deployment of vertices onto the best set of supervisors within an acceptable amount of time using the ABC algorithm, and it automatically redeploys vertices to improve application performance when the computing resources change and/or the user's needs cannot be met.

A supervisor continually listens to nimbus for vertices, and executes one or more vertices in one or many work processes. Each worker node can execute a limited number of work processes, which is determined by the number of available slots in the work node. The number of slots is determined by hardware capacities, such as CPU, memory. A monitor module is built into the supervisor, to monitor the inter-node and intra-node performance of the supervisor, such as the transmission latency and the remaining available slots. A monitor module is also built into the worker node, used to get the input and output data rate of each executor that runs on the worker node. The information kept by the monitor module is used for further vertex redeployment if necessary to better adapt to system changes.

Zookeeper, designed as a coordinator between nimbus and supervisors, stores the status of nimbus and supervisors, so zookeeper is stateful, while nimbus and supervisors are stateless.

5.2 ABC algorithm

ABC algorithm is a relatively new bio-inspired swarm intelligence algorithm, motivated by the intelligent foraging and the waggle dance behaviours of honeys bees swarm, proposed by Karaboga and Basturk (Saad et al., 2018), mainly for the purposes of continuous optimisation (Ma et al., 2019). ABC algorithm is better than genetic algorithm (GA) (Sadeghiram, 2017) and particle swarm optimisation (PSO) (Agarwal and Ranjan, 2019) over continuous space optimisation (Xiang et al., 2018).

In a honey bees swarm, there are three types of bees, which are employed bees, onlooker bees and scout bees. Employed bees are committed to searching available food sources in the neighbourhood, gathering honey information, and sharing honey information with onlooker bees. Onlooker bees select a relatively good food source according to the quality information of food sources, which

are provided by the employed bees, and then to further search a better food source with the application of local search. When a food source is exhausted by the employed and onlooker bees, the employed bees becomes a scout bee, and then search for a new food source randomly.

In the ABC algorithm, the numbers of employed bees or onlooker bees equals to the numbers of food source, which is denoted by *N*. Each food source represents a solution, so the terms 'food source' and 'solution' are hereby used interchangeably.

5.2.1 Initialisation

In the initialisation phase, the number of employed bees and onlooker bees are set to *N*. An initial population is generated randomly and consists of *N* solutions with *D*-dimensional variables. The i^{th} solution $X_i = \{x_{i1}, x_{i2}, \cdots, x_{iD}\}$ represents an element of the solution set. The j^{th} dimensional element of the i^{th} solution can be generated randomly by equation (7).

$$x_{ij} = x_j^{\min} + \left(x_j^{\max} - x_j^{\min}\right) \cdot r(0,1) \tag{7}$$

where $i \in \{1, 2, \cdots, N\}$, $j \in \{1, 2, \cdots, D\}$, $x_j^{\min}$ and $x_j^{\max}$ are the minimum and maximum of the j^{th} dimensional element, respectively, $r(0, 1)$ is a random real number in the interval [0, 1].

The fitness function $fit(X_i)$ of the i^{th} solution X_i can be calculated by equation (8).

$$fit(X_i) = \begin{cases} \dfrac{1}{1+f(X_i)}, \text{if} f(X_i) \geq 0 \\ 1+\left|f(X_i)\right|, \text{otherwise} \end{cases} \tag{8}$$

where $f(X_i)$ is the objective function value of the i^{th} solution X_i, determined by the response time of the deployment strategy.

5.2.2 Employed bees

In the employed bee phase, each employed bee begins to select a neighbour randomly and get a new food source V_i. The j^{th} dimensional element of the i^{th} new solution V_i can be generated randomly by equation (9).

$$v_{ij} = x_{ij} + \left(x_{ij} - x_{kj}\right) \cdot \phi_{ij}(-1,1), i \neq k \tag{9}$$

where $i, k \in \{1, 2, \cdots, N\}$, $j \in \{1, 2, \cdots, D\}$, X_k is a random selected neighbour of X_i, x_{kj} is the j^{th} dimensional element of the k^{th} solution X_k, $\phi_{ij}(-1, 1)$ is a random real number in the interval [–1, 1].

In order to keep v_{ij} within the limit of minimum and maximum of the j^{th} dimensional element, v_{ij} needs to be re-adjusted by equation (10).

$$v'_{ij} = \begin{cases} x_j^{\min}, \text{if } v_{ij} < x_j^{\min} \\ x_j^{\max}, \text{if } v_{ij} > x_j^{\max} \end{cases}. \tag{10}$$

If the fitness of the new solution V_i is better than that of the old solution X_i, the new solution V_i will be used to replace X_i; otherwise, the old solution X_i is retained.

5.2.3 Selection probability

The onlooker bee will select a food source according to the fitness of the food source. The selection probability p_i of the i^{th} solution X_i can be calculated by equation (11).

$$p_i = \frac{fit_i}{\sum_{j=1}^{SN} fit_j}. \tag{11}$$

Obviously, the larger the selection probability p_i, the more likely that the i^{th} solution X_i will be selected by the onlooker bees.

5.2.4 Onlooker bees

In the onlooker bees phase, when employed bees finish their search, they share the nectar amounts and the locations of solutions with the onlooker bees. Each onlooker bee begins to select a solution X_i according to its selection probability p_i, and selects a neighbour randomly and gets a new candidate solution V_i by equation (9). Similarly, if the fitness of the new candidate solution V_i is better than that of the selected solution X_i, the new candidate solution V_i will replace X_i; otherwise, the selected solution X_i will be retained.

The searching process continues until all the onlooker bees complete the similar search.

5.2.5 Scout bees

In the scout bees phase, if a solution X_i cannot be improved after repeated search, the solution X_i will be abandoned, and the employed bee becomes a scout bee and generates a new solution V_i to replace the solution X_i using equation (7).

5.3 Incremental online redeployment

If the latency or the throughput exceeds the user-defined limit, the running DAG needs to be redeployed online to improve the latency or throughput.

In the online redeployment phase, the redeployment solution can also be obtained using the modified ABC algorithm in Algorithm 1. To minimise the system fluctuation caused by the online redeployment, we compare the newest redeployment solution with the current deployment situation, and only incrementally redeploy some vertices that need to be adjusted urgently, rather than doing a full redeployment. The online redeployment algorithm is described in Algorithm 1.

The input of this algorithm includes the scheduling state of online DAGs and the current available capacity matrix $C_{v_{n\times m}}$. The output is the incremental online redeployment solution. Step 8 to Step 24 monitor the latency and throughput of each online DAGs, decide the timing of online redeployment, and incrementally redeploy vertices that need to be readjusted.

Algorithm 1 Incremental online redeployment algorithm

Input: scheduling state of online DAGs, current available capacity matrix $C_{v_{n\times m}}$ of computing nodes in a data centre.
Output: incremental online redeployment solution.
Initialise *num-candidate-DAG* = 0.
Set *limit* for redeployment threshold.
if DAG G or matrix $C_{v_{n\times m}}$ of computing nodes is null **then**
  Return null.
end if
for each DAG in the online DAGs **do**
Monitor the latency and throughput of each DAG G.
  if latency $l(G)$ of the DAG G is in $(l_{\min}(G), l_{\max}(G))$ or throughput $t(G)$ of the application is in $(t_{\min}(G), t_{\max}(G))$ **then**
    Set DAG G as the candidate redeployment application.
    num-candidate-DAG++.
  end if
  if *num-candidate-DAG* > *limit* or $l(G) > l_{\max}(G)$ or $t(G) > t_{\min}(G)$ **then**
    Get the best deployment solution by Algorithm 1.
      for each vertex in newest best redeployment solution and the current scheduling state **do**
        if a vertex v_i that is deployed in both deployment solutions but currently is deployed on computing nodes other than the ones in the newest redeployment solution, **then**
          Redeploy the v_i to the new computing nodes according the newest best redeployment solution.
        end if
      end for
    num-candidate-DAG = 0.
  end if
  Update resource state of each computing node in the data centre
end for
return incremental online redeployment solution.

6 Performance evaluation

This section focuses on evaluation of the proposed Pa-Stream, discussing the experimental environment and parameter settings, and providing an analysis on performance evaluation results.

6.1 Experimental environment

The proposed Pa-Stream system is implemented as an extension to Storm 1.0.2 (Toshniwal et al., 2014; Zhang et al., 2017). Stream application scenarios have been simulated on a computing cluster in Computer Architecture Laboratory at China University of Geosciences, Beijing. The cluster consists of 18 machines, with one designated node serving as master node, running Storm nimbus and UI. The computing node runs Linux CentOS 6.3 with dual 6-core, Intel Core (TM) i7-4790, 3.6 GHz, 8 GB memory, and 1 Gbps network interface cards. Three designated as the zookeeper node, each computing node runs Linux CentOS 6.3 with dual 6-core, Intel Core (TM) i7-4790, 3.6 GHz, 4 GB memory, and 1 Gbps network interface cards. The rest 14 machines work as the supervisor nodes, with each computing node runs Linux CentOS 6.3 with dual 4-core, Intel Core (TM) i5-8400, 2.8 GHz, 4 GB memory, and 1 Gbps network interface cards. Moreover, two streaming applications Top_N (Toshniwal et al., 2014) and WordCount (Toshniwal et al., 2014) are submitted to the cluster.

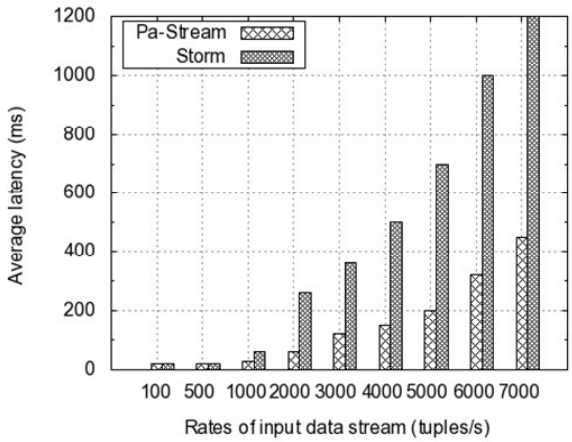
6.2 Performance results

The experimental setting contains three evaluation parameters: average latency *AL*, average throughput *AT*, and average computing node usage $u_{avg}(cn)$.

6.2.1 Average latency

Average latency *AL* or response time of a stream application is one of the most important performance indicators in distributed stream computing systems. On Storm platform, *AL* can be directly obtained through the Storm UI.

Figure 3 Average latency with different rates of inputs data stream



With the increase of rates of data stream, the average latency increases under both deployment strategies. For each input rate, Pa-Stream has a shorter average than the default Storm strategy.

As shown in Figure 3, when the rates are less than 1,000 tuples/s, both the average latency of Pa-Stream and the default Storm strategy all stay at a lower level. The difference between the two deployment strategies is small. When the rates climb to more than 1,000 tuples/s, the gap between the two strategies gets wider. The higher the data rate is, the more significant this difference is noted. When the rates increase to 7,000 tuples/s, the average latency of

Pa-Stream is gauged at 450 millisecond and the default Storm strategy goes to 1,200 millisecond.

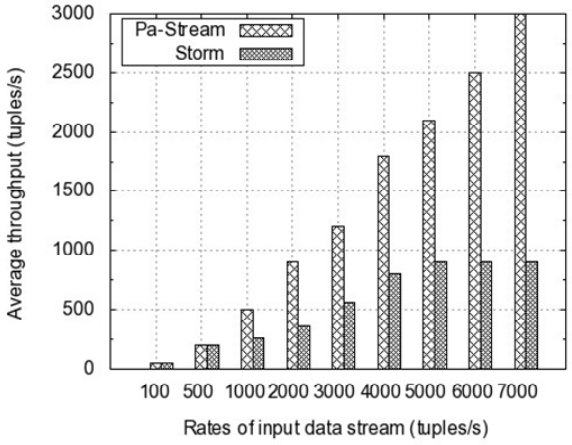
6.2.2 Average throughput

Average throughput *AT* reflects the overall processing ability of all running stream applications, evaluated by the number of output tuples per second per application.

With the increase of rates of data stream, the average throughput increases under both deployment strategies. At each rate, Pa-Stream beats the default Storm strategy with a higher average throughput.

As shown in Figure 4, with the increase of rates, the difference between the two strategies becomes obvious. The higher the data rate is, the more significant this difference is. When the rate reaches at 100 tuples/s, both the average throughput of Pa-Stream and the default Storm strategy behave at a similar level, gauged at 48 tuples/s and 49 tuples/s, respectively. The difference between the two deployment strategies is small. When the rate reaches at 7,000 tuples/s, the average throughput of Pa-Stream is gauged at 3,000 tuples/s, and that of the default Storm strategy goes to and 900 tuples/s. The difference is rather significant.

Figure 4 Average throughput with different rates of inputs data stream



7 Related work

In this section, we review two categories of related works: performance-aware deployment in distributed computing systems and ABC-based applications.

7.1 Performance-aware deployment in distributed computing systems

Application deployment plays an important role in distributed computing systems to help the application meet its functional and non-functional requirements. It is difficult to find an optimal schedule for a constraint-based DAG because of the fluctuating arrival rates of data stream and the varying amount of available computing resources. Performance-aware deployment is of crucial importance as the deployment state of applications directly affects the system performance (El-Kassabi et al., 2019).

In Saez et al. (2015), the authors focused on the persistence layer of applications. The effect of different deployment scenarios on application performance over time was investigated, and a performance-aware dynamic application (re)distributed design support process was proposed. The performance was evaluated with some well-known applications, such as the TPC-H benchmark.

In Kessler and Löwe (2012), the principles of a novel framework for performance-aware composition of sequential and explicitly parallel software components with variants implementation were described. Automatic composition results in a table-driven implementation that, for each parallel invocation of a performance-aware component, looked up the expected best implementation variant, processor allocation and schedule.

To summarise, performance-aware deployment in distributed computing systems has been studied in many works. However, few works have considered performance-ware deployment of streaming applications. In our work, we improve system performance by employing a performance-aware deployment strategy.

7.2 Bio-inspired-based applications in distributed systems

Bio-inspired algorithm can be applied in many fields (Cai et al., 2016). In Cai et al. (2019a), a multi-objective three-dimensional DV-hop localisation algorithm was proposed with NSGA-II. In Cui et al. (2019a), a CNNs and multi-objective algorithms were proposed for malicious code detection. In Cai et al. (2018), a bat algorithm with triangle-flipping strategy was proposed for numerical optimisation. ABC is a relatively new bio-inspired swarm intelligence algorithm, often resulting in better performance than GA, PSO over continuous space optimisation (Xiang et al., 2018).

For example, the topological design of a computer communication network is a well-known NP-hard problem. In Saad et al. (2018), a goal programming-based multi-objective artificial bee colony optimisation (MOABC) algorithm was proposed to solve the problem of topological design in distributed local area networks, and a comparative analysis was also done with regard to a non-dominated sorting GA and a Pareto-dominance PSO algorithm.

Virtual network embedding is also a hot research topic in the network virtualisation context. In Pathak et al. (2018), the authors proposed an approach based on ABC to address the dynamic virtual network embedding problem in a scenario with multiple infrastructure providers. A comparative study was conducted with other nature-inspired virtual network embedding algorithms. The findings affirmed that the proposed virtual network embedding approach performed well and produced better results.

In cloud manufacturing, it is possible that there are several conflicting criteria that need to be optimised simultaneously during the service composition and

selection. In this process, the trade-off regarding the quality of the composite services is a key issue in successful implementation of manufacturing tasks. In Zhou et al. (2018), an adaptive multi-population differential ABC algorithm was proposed for multiple-objective service composition in cloud manufacturing.

To summarise, the aforementioned application of ABC algorithm provides a valuable insight into the potential solutions for various problems in distributed systems. However, in big data stream computing systems, existing ABC algorithms cannot be applied to the stream application deployment. Particular challenges and opportunities of distributed stream computing system need to be considered, and some characteristics specific to data streams need to be considered when employing ABC algorithm to deploy stream applications in the distributed environments.

8 Conclusions and future work

To achieve low system latency, high system throughput, and high resource utilisation, a performance-aware deployment strategy should be able to determine when and how streaming applications are deployed according to the structure of applications and the available resources. It knows the available resources of each computing node, the dependencies between operators, and also the efficient deployment of inter-dependent operators to a best set of available computing nodes. The performance-aware deployment framework can process unbounded data streams in a scalable and efficient manner, minimising system latency and maximising system throughputs.

The paper makes the following contributions:

1 Investigated the performance-aware deployment of streaming applications over distributed and heterogeneous computing nodes, and provided a general system and application deployment model for distributed stream computing systems.

2 Proposed a streaming application deployment scheme by employing the artificial bee colony algorithm, and an incremental online redeployment strategy for running applications.

3 Developed and integrated the Pa-Stream into Apache Storm platform.

4 Implemented a prototype and tested the performance of the proposed Pa-Stream.

Our future work will be focusing on the following directions:

1 To develop a complete performance-aware deployment framework based on Pa-Stream.

2 To apply Pa-Stream in real big data stream computing application scenarios, such as real-time data monitoring scenario and real-time user portraits.

Acknowledgements

This work is supported by the National Natural Science Foundation of China under Grant No. 61972364; and the Fundamental Research Funds for the Central Universities under Grant No. 2652018081 and No. N181604015.

References

Agarwal, S. and Ranjan, P. (2019) 'TTPA: a two tiers PSO architecture for dimensionality reduction', *International Journal of Bio-Inspired Computation*, March, Vol. 13, No. 2, pp.119–130.

Cai, X., Gao, X. and Xue, Y. (2016) 'Improved bat algorithm with optimal forage strategy and random disturbance strategy', *International Journal of Bio-inspired Computation*, Vol. 8, No. 4, pp.205–214.

Cai, X., Wang, H., Cui, Z., Cai, J., Xue, Y. and Wang, L. (2018) 'Bat algorithm with triangle-flipping strategy for numerical optimization', *International Journal of Machine Learning and Cybernetics*, Vol. 9, No. 2, pp.199–215.

Cai, X., Wang, P., Du, L., Cui, Z., Zhang, W. and Chen, J. (2019a) 'Multi-objective three-dimensional DV-hop localization algorithm with NSGA-II', *IEEE Sensors Journal*, Vol. 19, No. 21, pp.10003–10015.

Cai, X., Zhang, J., Liang, H., Wang, L. and Wu, Q. (2019b) 'An ensemble bat algorithm for large-scale optimization', *International Journal of Machine Learning and Cybernetics*, DOI: 10.1007/s13042-019-01002-8.

Cardellini, V., Nardelli, M. and Luzi, D. (2016) 'Adaptive online scheduling in Storm', *Proceedings of the 2016 International Conference on High Performance Computing & Simulation, HPCS 2016*, IEEE Press, July, pp.583–590.

Cui, Z., Du, L. Wang, P., Cai, X. and Zhang, W. (2019a) 'Malicious code detection based on CNNs and multi-objective algorithm', *Journal of Parallel and Distributed Computing*, July, Vol. 129, pp.50–58.

Cui, Z., Zhang, J., Wang, Y., Cao, Y., Cai, X., Zhang, W. and Chen, J. (2019b) 'A pigeon-inspired optimization algorithm for many-objective optimization problems', *Science China Information Sciences*, July, Vol. 62, No. 7, pp.1–17.

Dias de Assunção, M., da Silva Veith, A. and Buyya, R. (2017) 'Distributed data stream processing and edge computing: a survey on resource elasticity and future directions', *Journal of Network and Computer Applications*, December, Vol. 103, pp.1–17.

Eidenbenz, R. and Locher, T. (2016) 'Task allocation for distributed stream processing', *Proceedings of 35th Annual IEEE International Conference on Computer Communications, INFO-COM 2016*, IEEE Press, July, Article no. 7524433.

El-Kassabi, H.T., Serhani, M.A., Dssouli, R. and Navaz, A.N. (2019) 'Trust enforcement through self-adapting cloud workflow orchestration', *Future Generation Computer Systems*, August, Vol. 97, pp.462–481.

Hirzel, M., Soulé, R., Schneider, S., Gedik, B. and Grimm, R. (2014) 'A catalog of stream processing optimizations', *ACM Computing Surveys*, April, Vol. 46, No. 4, pp.1–34.

Kessler, C. and Löwe, W. (2012) 'Optimized composition of performance-aware parallel components', *Concurrency Computation Practice and Experience*, April, Vol. 24, No. 5, pp.481–498.

Kotto-Kombi, R., Lumineau, N. and Lamarre, P. (2017) 'A preventive auto-parallelization approach for elastic stream processing', *Proceedings of the 37th International Conference on Distributed Computing Systems, ICDCS 2017*, IEEE Press, June, pp.1532–1542.

Li, C., Zhang, J. and Luo, Y. (2017) 'Real-time scheduling based on optimized topology and communication traffic in distributed real-time computation platform of storm', *Journal of Network and Computer Applications*, June, Vol. 87, pp.100–115.

Li, T., Tang, J. and Xu, J. (2016) 'Performance modeling and predictive scheduling for distributed stream data processing', *IEEE Transactions on Big Data*, December, Vol. 2, No. 4, pp.353–364.

Lombardi, F., Aniello, L., Bonomi, S. and Querzoni, L. (2018) 'Elastic symbiotic scaling of operators and resources in stream processing systems', *IEEE Transactions on Parallel and Distributed Systems*, March, Vol. 29, No. 3, pp.572–585.

Ma, L.B., Wang, X.W., Shen, H. and Huang, M. (2019) 'A novel artificial bee colony optimiser with dynamic population size for multi-level threshold image segmentation', *International Journal of Bio-Inspired Computation*, January, Vol. 13, No. 1, pp.32–44.

Mencagli, G., Torquati, M. and Danelutto, M. (2018) 'Elastic-PPQ: a two-level autonomic system for spatial preference query processing over dynamic data streams', *Future Generation Computer Systems*, February, Vol. 79, pp.862–877.

Pathak, I., Tripathi, A. and Vidyarthi, D.P. (2018) 'A model for virtual network embedding using artificial bee colony', *International Journal of Communication Systems*, July, Vol. 31, No. 10, pp.1–22, Article number e3573.

Saad, A., Khan, S.A. and Mahmood, A. (2018) 'A multi-objective evolutionary artificial bee colony algorithm for optimizing network topology design', *Swarm and Evolutionary Computation*, February, Vol. 38, pp.187–201.

Sadeghiram, S. (2017) 'Bacterial foraging optimisation algorithm, particle swarm optimisation and genetic algorithm: a comparative study', *International Journal of Bio-Inspired Computation*, May, Vol. 10, No. 4, pp.275–282.

Saez, S.G., Andrikopoulos, V., Leymann, F. and Strauch, S. (2015) 'Design support for performance aware dynamic application (re-)distribution in the cloud', *IEEE Transactions on Services Computing*, March–April, Vol. 8, No. 2, pp.225–239.

Thoman, P., Dichev, K., Heller, T., Iakymchuk, R., Aguilar, X., Hasanov, K., Gschwandtner, P., Lemarinier, P., Markidis, S., Jordan, H., Fahringer, T., Katrinis, K., Laure, E. and Nikolopoulos, D.S. (2018) 'A taxonomy of task-based parallel programming technologies for high-performance computing', *The Journal of Supercomputing*, April, Vol. 74, No. 4, pp.1422–1434.

Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S. and Ryaboy, D. (2014) 'Storm@twitter', *Proceedings of 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD 2014*, ACM Press, June, pp.147–156.

Wang, C., Meng, X., Guo, Q., Weng, Z. and C. Yang. (2017) 'Automating characterization deployment in distributed data stream management systems', IEEE Transactions on Knowledge and Data Engineering, December, Vol. 29, No. 12, pp. 2669-2681.

Xiang, W., Meng, X., Li, Y., He, R. and An, M. (2018) 'An improved artificial bee colony algorithm based on the gravity model', *Information Sciences*, March, Vol. 429, pp.49–71.

Zhang, J., Li, C., Zhu, L. and Liu, Y. (2017) 'The real-time scheduling strategy based on traffic and load balancing in storm', *Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications, HPCC 2016*, IEEE Press, January, pp.372–379.

Zhou, J., Yao, X., Lin, Y., Chan, F.T.S. and Li, Y. (2018) 'An adaptive multi-population differential artificial bee colony algorithm for many-objective service composition in cloud manufacturing', *Information Sciences*, August, Vol. 456, pp.50–82.