Towards making big data applications network-aware in edge-cloud systems

Dávid Haja*†, Balázs Vass†, László Toka*‡
*MTA-BME Network Softwarization Research Group, †Budapest University of Technology and Economics,
‡MTA-BME Information Systems Research Group

***Abstract*—The amount of data collected in various IT systems has grown exponentially in the recent years. So the challenge rises how we can process those huge datasets with the fulfillment of strict time criteria and of effective resource consumption, usually posed by the service consumers. This problem is not yet resolved with the appearance of edge computing as wide-area networking and all its well-known issues come into play and affect the performance of the applications scheduled in a hybrid edge-cloud infrastructure. In this paper, we present the steps we made towards network-aware big data task scheduling over such distributed systems. We propose different resource orchestration algorithms for two potential challenges we identify related to network resources of a geographically distributed topology: decreasing end-to-end latency and effectively allocating network bandwidth. The heuristic algorithms we propose provide better big data application performance compared to the default methods. We implement our solutions in our simulation environment and show the improved quality of big data applications.**

***Index Terms*—Big data, resource orchestration, network latency, bandwidth, geo-distributed network topology**

I. Introduction

In the recent years two novel concepts appeared which extends traditional cloud computing by deploying compute resources closer to customers and end devices in terms of network latency: edge-cloud computing [1] and mobile edge computing [2]. These approaches, closely integrated with carrier-networks, enable several future 5G and beyond applications and network services, such as novel Industry 4.0 use-cases, Tactile Internet, remote driving or extended reality. This extension creates the opportunity for: i) enabling latency-critical communication between the actors which is required by various envisioned services; ii) customers' devices can offload computational tasks to this distributed environment instead of consuming their local resources.

Data analysis is one of the most time-critical services. Today's resource orchestration algorithms, used for allocating system resources to big data application instances, are usually designed for a single data center. Since latency, and bandwidth-capacity rarely pose problems in the typically dense data center networks, the common solutions do not take into account the characteristics of the underlying network. In contrast, these network aspects can easily cause application performance degradation when the network topology contains edge nodes inter-connected with wide-area networking. First, network latency worsens the end-to-end processing time in stream analytics. Second, if bandwidth capacity is scarce, which is usually the case in edge uplinks, data-intensive analytic applications will suffer serious performance degradation. We argue that these two networking aspects are important for a big data system designer, particularly in a geographically distributed edge-cloud scenario.

In this work, we engage in both of these problem domains and propose various orchestration methods that can significantly improve the system performance compared to the default scheduling baseline. Our contribution is two-fold: i) task scheduling in map-reduce type processing frameworks, which results in 30% lower end-to-end completion time on average than the baseline of Spark; ii) resource orchestration that yields more effective bandwidth utilization than what is obtained with a default YARN (Yet Another Resource Negotiator) operation [3]. Our general finding is that the widely used open-source big data resource orchestrators are not yet ready for hybrid edge-cloud deployments where the quality of wide-area networks might leave a huge mark on the application performance. Our work is a pioneer step towards geo-distributed network-aware orchestration of such systems: to the best of our knowledge, these aspects have never been investigated in relation to big data and edge-cloud infrastructure.

The paper is organized as follows. In Sec. II we show how we suggest to place *mapper* and *reducer* tasks in order to decrease the network's negative effect in the end-to-end job completion time. As our other contribution, we provide a bandwidth-aware resource orchestration method in Sec. III. At the end of the paper, we present the related work in Sec. IV and our conclusions in Sec. V.

II. Delay-aware scheduling of MapReduce tasks

In this section, we show how we can speed up stream analytics jobs containing mapper and reducer tasks by placing task executors with the minimum possible network latency among them. We prove that this problem is hard, and we suggest fast heuristics to solve it, and we show how our solution performs compared to the industry-standard baseline.

A. Execution time model with network parameters

Spark uses tasks to read and to process data in a cluster or even across clusters. Spark relies on the location of the data, so data locality can affect Spark applications performance.

Some of the tasks contain resource requests which specify the servers they prefer to run on, i.e., those that contain the data portions need to be processed. The Spark framework is designed to ensure that these tasks will be initiated on one of the requested servers if they have free computing resources. Otherwise, the algorithm attempts to run the task on another server in the same rack. If this is not possible, then the algorithm randomly selects a server with available resources from the entire topology to run the task. Selecting randomly a server does not significantly affect the application performance in a single data center environment regarding the network delay. On the other hand, this selection policy might have a huge impact on a geographically distributed system, where the network latency can be high. Our proposed solution overcome these shortcomings in a geographically distributed topology: the purpose of our algorithm is to reduce the execution time of streaming analytics applications in an edge-cloud system by minimizing network delays between the components.

We model the edge-cloud topology with a graph in which a vertex is one server, switch or gateway. The servers are grouped into server racks which are further grouped into clusters. We distinguish two types of clusters: data centers and edge clusters. Each cluster contains at least one gateway node, which provides the connection between the cluster and the Internet. The nodes of the graph are connected by undirected edges, which correspond to physical links. In this case, edges are weighted with the measured average delay between their endpoints. Servers within a cluster form a clique, and the gateways of all clusters also form a full-mesh network. Each server's capacity is described by a positive integer, which defines the number of tasks that can be placed on the server; we assume that each task requires the same amount of compute resource. Edge clusters have less compute resources than data centers. An example of this topology is drawn in Fig. 1.

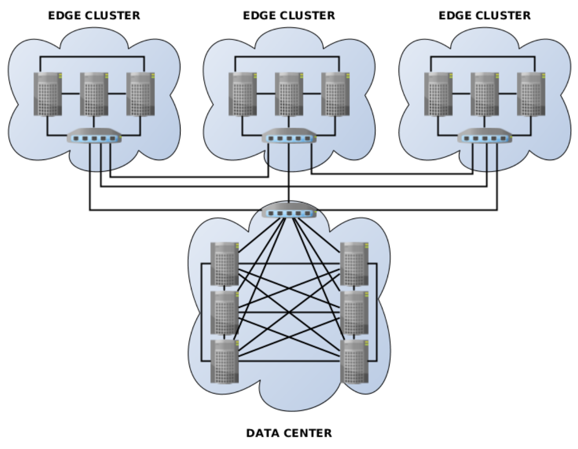

Fig. 1: Topology graph for the delay-aware scheduling solution

The submitted Spark jobs are also represented as graphs, where a vertex is a task of the job and the edges show the dependency between the tasks defined by the intermediate data flows. The challenge here is to create an algorithm, which places the map and reduce tasks in the topology so that the maximum of the link delays from the location of the map tasks to the reduce tasks' location may be as low as possible.

Lemma 1: Minimizing the delay between map and reduce tasks to be scheduled is an NP-complete problem.

Proof: The job graph is a complete bipartite graph where one vertex set contains the map tasks and the other set contains the reduce tasks. Let us create a complete graph G which shows a logical abstraction of the topology graph. Every vertex in G matches the nodes in the topology graph and has a capacity value. Every edge in G represents a logical connection between the two endpoints weighted with the sum of delays found on the shortest path links between the respective topology vertices. Our task is to find a subgraph in G which is equal with the graph (G_y) where the job graph components are placed. Clearly, this problem is in NP. It is easy to see that this subgraph is also a complete bipartite graph as the application graph. Now let us construct subgraphs G_x from G, which contains the first x edges with the lowest delays ($x \in \{1, \dots, |E(G)|\}$). Deciding whether G_y is part of G_x contains the following special case: deciding whether the complete balanced bipartite graph $K_{k,k}$ is a subgraph of G_x, which problem is NP-complete (GT24 in [4]). A complete bipartite graph is balanced if the two subsets of vertices have equal cardinality. Since the set of bipartite graphs is a subset of the general graphs, looking for a balanced complete bipartite graph in a general graph is also NP-complete. This means that deciding whether G_y is a subgraph of G_x is NP-complete (for an arbitrary x), and thus our original minimization problem is NP-complete, too. ■

The problem is formalized as an ILP in Fig. 2. One can easily check that the solutions of the ILP are exactly the solutions of the depicted problem as follows. Eq. (1) assures that as a result of the scheduling, each task is assigned to exactly one physical node. Due to Eq. (2), the total amount of resources required by the tasks mapped to a given node cannot exceed the resources available at the node. The flow constraints are given by Eq. (3). By Eq. (4) and target function (5), the optimal d is the smallest possible value of the maximum of the sum of the delay values on the paths from the physical locations of the map tasks to the reduce tasks.

B. Our latency-aware task placement algorithm

We propose a heuristic algorithm to solve the hard problem defined in Sec. II-A. The pseudocode of our proposed algorithm is shown in Alg. 1. When our algorithm processes a job scheduling request, it iterates through all the job's tasks and places them in the topology. First, the algorithm deploys the map tasks examining the following cases in sequence:

- Place the map task on one of the locality constraint servers if it has available compute resources.
- If none of the locality constraint servers have sufficient resources, the algorithm tries to start the task on a node with available resources in the same racks.
- If none of the racks with the locality constraint servers have available servers, then we calculate the delays from

Notation	Description
$V_s(V_m, V_r), E_s$	vertices and edges of the job graph, where V_m notates the map and V_r notates the reduce components
V_t, E_t	vertices and edges of the topology graph
$n \in V_t$	"Internet" node in topology
$x_u^i : i \in V_s, u \in V_t$	1 if task i placed on vertex u else 0
$y_{u,v}^{i,j} : (i,j) \in E_s, (u,v) \in E_t$	1 if job graph edge (i,j) contains physical path (u,v) else 0
$r_i \in \mathbb{N}$	task i's resource requirement
$\rho_u \in \mathbb{N}$	server u's available resources
$\delta_{u,v} \in \mathbb{N}$	delay on physical link (u,v)
$\chi_{u,v} \in \mathbb{N}$	bandwidth capacity of physical link (u,v)
$\beta_{i,j} \in \mathbb{N}$	bandwidth requirement on virtual link (i,j)

$$\forall i \in V_s : \sum_{u \in V_t} x_u^i = 1 \tag{1}$$

$$\forall u \in V_t : \sum_{i \in V_s} x_u^i r_i \leq \rho_u \tag{2}$$

$$\forall (i,j) \in E_s, \forall u \in V_t :$$
$$\sum_{v:(u \to v) \in V_t} y_{u,v}^{i,j} - \sum_{w:(w \to u) \in V_t} y_{w,u}^{i,j} = x_u^i - x_u^j \tag{3}$$

$$\forall (i,j) \in E_s : d \geq \sum_{(u,v) \in E_t} y_{u,v}^{i,r} \delta_{u,v} \tag{4}$$

$$\min d \tag{5}$$

Fig. 2: ILP formulation for delay-optimal map-reduce placement

the constraint servers to the least utilized servers of each cluster and select the one with the smallest delay.

Algorithm 1 Delay-aware task placement

```
for each $task \in request.map\_tasks + request.reduce\_tasks$ do
    if $\exists task.locality\_constraint$ then
        for each $constraint\_server \in locality\_constraint$ do
            if $constraint\_server.has\_available\_resource()$ then
                $task.place(constraint\_server)$
                $break$
            end if
        end for
        if $\neg task.placed$ then
            for each $constraint\_server \in locality\_constraint$ do
                $rack \leftarrow constraint\_server.rack$
                if $rack.has\_available\_resource()$ then
                    $task.place(rack.least\_utilized\_server())$
                    $break$
                end if
            end for
        end if
        if $\neg task.placed$ then
            $min\_delay \leftarrow \infty$
            $available\_servers \leftarrow least\_utilized\_servers(\forall clusters)$
            for each $server \in available\_servers$ do
                for each $constraint\_server \in locality\_constraint$ do
                    $delay \leftarrow delay(server, constraint\_server)$
                    if $delay < min\_delay$ then
                        $min\_delay \leftarrow delay$
                        $chosen\_server \leftarrow server$
                    end if
                end for
            end for
            $task.place(chosen\_server)$
        end if
    else
        $min\_delay \leftarrow \infty$
        $available\_servers \leftarrow least\_utilized\_servers(\forall clusters)$
        for each $server \in available\_servers$ do
            $delay \leftarrow max\_delay\_between\_nodes(server, map\_places)$
            if $max\_delay < min\_delay$ then
                $min\_delay \leftarrow max\_delay$
                $chosen\_server \leftarrow server$
            end if
            $task.place(chosen\_server)$
        end for
    end if
end for
```

After all map tasks are placed, the algorithm schedules the reduce tasks with the goal of placing them as close to the map tasks as possible regarding network delay. To achieve this, first, the least utilized servers from each cluster are listed. Then, by iterating through this list, the maximum delay between the actual server and all the servers where map tasks are placed on is recorded. Finally, the server with the minimum recorded delay is chosen to host the reduce task.

Our algorithm requires an initial phase, in which the delay between each node pair is measured. The complexity of this phase is $\mathcal{O}(V^2)$ where V is the number of servers. We need to rerun this phase only when the topology changes. The complexity of creating a list with the least utilized servers from each cluster is $\mathcal{O}(V)$. The length of this list equals the number of clusters in our topology, denoted by C. Let us denote the number of the locality constraint servers belonging to a map task with K. Finding the shortest path between two vertices can be done with Dijkstra's algorithm, with the worst case performance of $\mathcal{O}(E + V \log V)$, where E is the number of edges in the graph. So the complexity of deploying one map task can be estimated with $\mathcal{O}(V + CK(E + V \log V)))$. Let us denote the number of tasks in a job with T. Thus the worst case complexity of the delay-aware task placement algorithm is $\mathcal{O}(V^2 + T(V + CK(E + V \log V)))$, which is, in case of $C, K, T \ll V$, $\mathcal{O}(V^2)$.

C. Numerical analysis

We implemented a simulation environment, where we compared our proposed algorithm with Spark's task placement algorithm. In the simulations we used one topology with 50 edge clusters and 5 data centers. In Table I we show the delay values we used in our simulations.

TABLE I: Network parameters in our simulations

Link type	Delay [ms]	Bandwidth [Gbps]
server-server (in data center, the same rack)	1	100
server-server (in edge cluster, the same rack)	1	50
rack-rack (in the same cluster)	5	10
server-gateway (in the same cluster)	5	10
gateway-gateway (between two cluster)	20-200	-
data center gateway uplink	-	5
edge cluster gateway uplink	-	1

In the simulations all jobs contained five map and two reduce tasks. All map tasks contained three locality constraint servers, where the task was preferred to be placed. We run 5 scenarios which differed in the locality constraints for map tasks, i.e., locality constraint servers were randomly picked from 1, 2, 3, 4, or all 5 edge clusters respectively. Both Spark's and our scheduling algorithm solution received the same set of job requests during the simulations. We compared the achieved delays between the map and the reduce tasks.

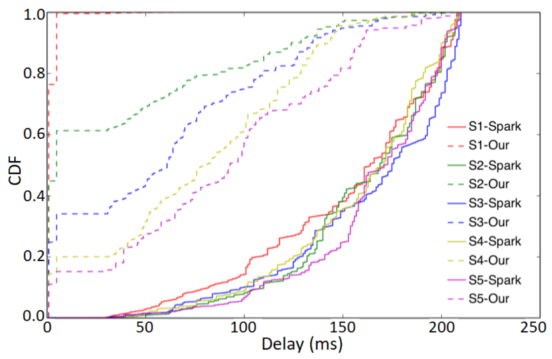

Fig. 3: Delay-aware simulation results

In Fig. 3 we plot the CDF of delays achieved by the two scheduling methods in all the scenarios. On the x-axis delay is shown in milliseconds. Our algorithm showed better performance than Spark in all scenarios. However, the performance of our algorithm drops with the increase of the number of clusters that can be selected for locality constraints, i.e., the achieved delay between the map and the reduce tasks are increasing. This is an expected phenomenon, since the higher number of clusters we can put our map tasks in, the more likely will we have data flows crossing over the inter-cluster network where links have high delay values. In the meantime increasing the number of clusters from where locality constraints can be picked does not have much effect on Spark's performance. This is because Spark does not take into account delay, therefore randomly selecting servers to reduce tasks is not affected by the number of locality constraints.

III. Bandwidth-aware executor placement for scarce WAN capacity

In this section we show that efficient bandwidth allocation can lead to better performance in big data processing. We reach high performance by designing our orchestration method with edge-cloud topologies in mind. First we show that the bandwidth-aware scheduling problem is hard, then in order to solve the problem we propose a greedy heuristic algorithm, finally, we compare its performance with that of YARN.

A. Bandwidth-aware executor placement problem

For many big data applications YARN deploys their executors that process data. These executors may have requirements dictated by the location of their input. These requirements are translated into resource requests during the scheduling process inside the ResourceManager, the main component of YARN. Resource requests include the following parameters: preferred resources, number of containers per resource, locality preference and priority. Since the ResourceManager has a global view of the available infrastructure resources, it is able to pinpoint the servers with available resources where the individual executors can be placed. The data locality strategy of YARN is similar to the strategy of Spark, presented in II-A. The key of our solution lies within selecting servers with awareness of network bandwidth capacities.

We use a similar topology graph for modeling the bandwidth-aware executor placement problem as the one described in Sec. II-A. There are two points in which the model differs though: i) the weight on an edge represents the available bandwidth between its two endpoints; ii) cluster gateways are connected to a central vertex that represents the Internet. We assume that a data center's connection to the Internet has larger bandwidth than an edge cluster's. We model job requests based on what they are characterized in practice: an application has an application master that requires executors in the big data system; among these executors some have locality constraints. Our goal is to place executors to such servers that the intermediate data flow among them may enjoy the highest available bandwidth. To achieve this goal, we propose applying a heuristic algorithm that we present in Sec. III-B.

Lemma 2: The bandwidth-aware executor placement problem (BAEP) is NP-complete.

Proof: Based on the ILP formulation of the BAEP, the problem is in NP. To prove the NP-hardness of BAEP, we show that it is at least as hard as the bin packing, which is an NP-complete problem [4, SR1]. The bin packing problem is the following. *Instance:* set U of items, a size $s(u)$ for each $u \in U$, a positive integer bin capacity C, and a positive integer k; *Question:* is there a partition of U into disjoint sets $U_1, \ldots, U_k$ such that the sum of the sizes in U_i is C or less?

Each instance of the bin packing can be transformed via a function f into a BAEP instance the following way. Let $f(U) = V_s$, $f(s(i \in U)) = r_{i \in V_s}$, $|V_t| = k$ with $\rho(u) = C$ for all $u \in V_t$, $\delta_{u,v} = 0$ and $\chi_{u,v} = +\infty$ for all $(u, v) \in E_t$, and finally $\beta_{i,j} \in \mathbb{N}^+$ is arbitrary for all $(i, j) \in E_s$. It can be seen that the resulting BAEP instance has a feasible solution exactly if the original bin packing problem instance is solvable. Thus, the BAEP problem is NP-hard, since it is at least as hard as the bin packing. The proof follows. ■

Similarly to the delay-aware scheduling in Sec. II, we formalize the problem as an ILP. The notations of our bandwidth-aware executor placement problem can be found in Fig. 2. Conditions (1), (2), and (3) of Sec. II are also applied here. An extra condition is defined for our bandwidth-aware executor placement problem, which states that the total bandwidth of virtual links mapped to the same physical link in the WAN network cannot be greater than the bandwidth capacity of the link, formalized in (6). The target function of the bandwidth-aware optimization is to minimize the allocated bandwidth in the WAN network between all map and reduce executor pairs, formalized in (7). The ILP for the bandwidth-aware executor placement problem can be completed as follows:

$$\forall (u, n) \in E_t : \sum_{(i,j) \in E_s} y^{i,j}_{u,n} \beta_{i,j} \leq \chi_{u,n} \qquad (6)$$

$$\min : \sum_{\substack{(i,j)\in E_s \\ (u,n)\in E_t}} y_{u,n}^{i,j}\beta_{i,j} \tag{7}$$

B. Our heuristic algorithm for executor placement

The purpose of our algorithm is to place the executors to achieve the maximum network bandwidth for the intermediate data flows; the pseudo-code is presented in Alg. 2.

Algorithm 2 Executor placement with bandwidth maximization for intermediate data flows

```
for each $executor \in request.executors$ do
    $max\_bw \leftarrow 0$
    if $\exists executor.locality\_constraint$ then
        for each $constraint\_server \in locality\_constraint$ do
            if $constraint\_server.has\_available\_resource()$ then
                $executor.place(constraint\_server)$
                $break$
            end if
        end for
        if $\neg executor.placed$ then
            for each $constraint\_server \in locality\_constraint$ do
                $rack \leftarrow constraint\_server.rack$
                if $rack.has\_available\_resource()$ then
                    $executor.place(rack.least\_utilized\_server())$
                    $break$
                end if
            end for
        end if
        if $\neg executor.placed$ then
            $available\_servers \leftarrow least\_utilized\_servers(\forall clusters)$
            for each $server \in available\_servers$ do
                for each $constraint\_server \in locality\_constraint$ do
                    $bw \leftarrow min\_bw(server, constraint\_server)$
                    if $bw > max\_bw$ then
                        $max\_bw \leftarrow bw$
                        $chosen\_server \leftarrow server$
                    end if
                end for
            end for
            $executor.place(chosen\_server)$
        end if
    else
        $reduce\_executor \leftarrow executor$
        $available\_servers \leftarrow least\_utilized\_servers(\forall clusters)$
        for each $server \in available\_servers$ do
            $bw \leftarrow min\_bw(server, map\_executor\_hosts)$
            if $bw > max\_bw$ then
                $max\_bw \leftarrow bw$
                $chosen\_server \leftarrow server$
            end if
            $reduce\_executor.place(chosen\_server)$
        end for
    end if
end for
```

Our algorithm first deploys the application master on a randomly selected server, then it places the executors in the topology one by one. First, the algorithm deploys the map executors: the map executor placement strategy is the same as in Sec. II-B, except that if none of the racks with the locality constraint servers have available servers, then the algorithm calculates the available bandwidth from the constraint servers to the least utilized servers of each cluster and selects the server with the maximum bandwidth.

After the map executor placement, the reduce executors are deployed. The algorithm maximizes bandwidth between map and reduce executors. First, the least utilized servers from each cluster are listed, then the algorithm iterates through this list and records the minimum bandwidth between the actual server and all the servers where map executors are placed at. Finally, the server with the maximum calculated bandwidth is chosen for hosting the reduce executor.

The complexity of this bandwidth-aware heuristic algorithm is equal with the delay-aware algorithm's complexity presented in Sec. II-B.

C. Simulation settings and results

In order to compare our algorithm with the algorithm used by YARN in a geographically distributed topology, we developed our own simulator. We applied the model proposed in Sec. III-A. We ran both YARN's and our algorithm on the same topology with 50 edge clusters and 5 data centers. The bandwidth values used in the simulations are shown in Table I.

We investigated various cases where the number of map executors in an application ranges from 2 to 8. We set the total number of executors, i.e., map and reduce, to 10. In terms of locality constraints, we applied the same diversity of application examples as in the simulations presented in Sec. II-C. After deploying an application, if it has a flow between a map and a reduce executor passing through the WAN network, we reduce the available bandwidth of WAN links of the respective clusters by 10Mbps. Each simulation lasted until all the computing resources or the available bandwidth to a cluster had been exhausted. At that point, we recorded the number of deployed executors and the number of flows between clusters generated by each application.

The results show that application requests processed by our algorithm used less WAN network bandwidth for forwarding data, so it is not surprising that we could place more executors in the topology than YARN's algorithm. Fig. 4 compares the performance of YARN's algorithm with that of our algorithm, in terms of the number of successfully deployed executors. Each bar group shows the results of a case with a specific number of map executors in each application, i.e., 2, 3, etc. In each simulation the locality constraint for map executors were defined as a randomly picked edge cluster. The y-axis shows the number of the placed executors at the end of the simulations.

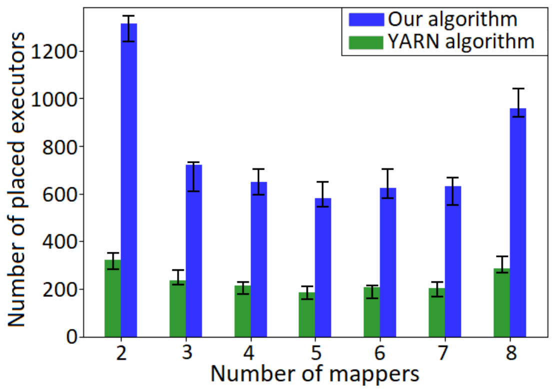

Fig. 4: WAN flow-aware executor placement simulation results

Our algorithm yielded better results than YARN in all scenarios: it could deploy, on average, 4-times more executors before running out of WAN bandwidth compared to YARN. By increasing the number of map executors per application, the performance of our algorithms drops, but after a certain point it increases again. This is expected behavior: if all executors are placed in different clusters, the number of WAN flows will be equal to $2xy$, where x is the number of map and y is the number of reduce executors, because for each WAN flow we need to decrease the available bandwidth on the uplink of the map, and that of the downlink of the reduce executor's cluster. As $x + y$ is set to 10, the maximum number of WAN flows is reached when $x = y = 5$, hence the relatively low number of successfully deployed applications at $x = 5$ in Fig. 4.

IV. Related Work

Virtual function placement: Network delay affects the speed of big data applications. Optimal big data deployment can be formulated as placing virtual nodes or functions in a physical topology by considering network resources such as delay. VNE (Virtual Network Embedding) [5] gives a general description of this problem. Numerous results [6]–[9] have been proposed in recent years. However, we see that placing executors, tasks of big data systems have not been examined regarding network delay.

MapReduce data analysis in hybrid cloud: MapReduce application placement solutions working in geographically distributed environment are proposed in [10]–[13]. They show improvement regarding the application performance in distributed, multiple data center context. The authors of these papers suggest moving the input data for the MapReduce applications. Moving input data can be inefficient in such an edge-cloud topology, so our algorithms do not rearrange input data for the data analysis applications.

Spark task scheduling in geo-distributed topologies: Authors of [14]–[18] examine how Spark tasks can be placed optimally in geo-distributed environments. Iridium [14] does not consider limitations in compute and storage in the clusters. In contrast, our solution considers computational capacities. G. Zhang et al. in [15] excludes the network latency from the execution time calculation. Solutions in [16]–[18] articles calculate with the intermediate data size along with which they define their task placement cost. In our work, we do not alter intermediate task sizes.

Big data application scheduling in SDN compatible data centers: In [19], [20] the authors attempt to improve network utilization in SDN-compatible data centers. The main difference between their research and ours is that our models can be applied in legacy networks.

V. Conclusion and future work

In the presented research we investigate the performance degradation of big data applications once they are deployed in a geographically distributed infrastructure. As edge computing becomes rather the norm than the exception nowadays, the long-researched networking aspects play an important role when it comes to the quality of data analysis frameworks. We showed that the default scheduling and resource orchestration strategies of the mainstream big data ecosystems are not prepared to cope with the challenges the wide-area networks pose. Realizing this, we made pioneer steps and proposed fast topology-aware algorithms for Spark, MapReduce, and YARN in order to improve end-to-end job completion and bandwidth utilization respectively. We have shown in numerical simulations that greedy heuristics yield significant amelioration compared to the default baseline in those aspects. There is room for advancement: we plan to extend the discussed open source resource orchestrator solutions with the proposed algorithms and share the code with the respective Apache projects along with the extension that manually receives or automatically discovers the underlying topology. The theoretical way forward is to further polish the algorithms, particularly on the basis of feedback from the real-world deployments' underlying topologies.

Acknowledgment

This work was supported by the National Research, Development and Innovation Office (NKFIH) under the research and development project in Hungarian-Korean cooperation (project identifier: 2018-2.1.17-TT-KR-2018-00012).

References

[1] C. Byers, "Architectural imperatives for fog computing: Use cases, requirements, and architectural techniques for fog-enabled iot networks," *IEEE Communications Magazine*, 2017.

[2] P. Mach *et al.*, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Communications Surveys & Tutorials*, 2017.

[3] V. Vavilapalli *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *ACM SoCC*, 2013.

[4] D. Johnson *et al.*, *Computers and intractability: A guide to the theory of NP-completeness*. WH Freeman San Francisco, 1979.

[5] A. Fischer *et al.*, "Virtual network embedding: A survey," *IEEE Communications Surveys & Tutorials*, 2013.

[6] M. Yu *et al.*, "Rethinking virtual network embedding: substrate support for path splitting and migration," *ACM SIGCOMM*, 2008.

[7] N. Chowdhury *et al.*, "Virtual network embedding with coordinated node and link mapping," in *IEEE INFOCOM*, 2009.

[8] D. Haja *et al.*, "How to orchestrate a distributed openstack," in *IEEE INFOCOM*, 2018.

[9] B. Németh *et al.*, "Fast and efficient network service embedding method with adaptive offloading to the edge," in *IEEE INFOCOM*, 2018.

[10] A. Ruiz-Alvarez *et al.*, "Toward optimal resource provisioning for cloud mapreduce and hybrid cloud applications," in *IEEE/ACM BDCAT*, 2014.

[11] M. Cavallo, "H2f: a hierarchical hadoop framework to process big data in geo-distributed contexts," 2018.

[12] B. Heintz *et al.*, "End-to-end optimization for geo-distributed mapreduce," *IEEE Transactions on Cloud Computing*, 2014.

[13] Q. Zhang *et al.*, "Improving hadoop service provisioning in a geographically distributed cloud," in *IEEE CLOUD*, 2014.

[14] Q. Pu *et al.*, "Low latency geo-distributed data analytics," *ACM SIGCOMM*, 2015.

[15] G. Zhang *et al.*, "Improving performance for geo-distributed data process in wide-area," in *IEEE CIT*, 2017.

[16] Z. Hu *et al.*, "Flutter: Scheduling tasks closer to data across geo-distributed datacenters," in *IEEE INFOCOM*. IEEE, 2016.

[17] B. Cheng *et al.*, "Geelytics: Geo-distributed edge analytics for large scale iot systems based on dynamic topology," in *IEEE WF-IoT*, 2015.

[18] L. .Gu *et al.*, "A general communication cost optimization framework for big data stream processing in geo-distributed data centers," *IEEE Transactions on Computers*, 2015.

[19] G. Wang *et al.*, "Programming your network at run-time for big data applications," in *ACM HotSDN*, 2012.

[20] P. Qin *et al.*, "Bandwidth-aware scheduling with sdn in hadoop: A new trend for big data," *IEEE Systems Journal*, 2015.