An Innovative MapReduce-Based Approach of Dijkstra's Algorithm for SDN Routing in Hybrid Cloud, Edge and IoT Scenarios

Alina Buzachis, Antonino Galletta, Antonio Celesti(✉), and Massimo Villari

MIFT Department, University of Messina, Messina, Italy
{abuzachis,angalletta,acelesti,mvillari}@unime.it

Abstract. Nowadays, with the advent of Cloud/Edge Computing and Internet of Things (IoT) technologies, we are facing with a tremendous increase of network connections required by different new cutting-edge distributed applications spread over a wide geographical area. Specifically, the proliferation of IoT devices used by such applications and associated data streams require a highly dynamic network ecosystem; the traditional network technologies are not adequate to efficiently support them in terms of routing strategies. In order to deploy such applications, providers need an advanced awareness of the Cloud/Edge and IoT networks in terms of flexible packets routing that can compute the paths according to different parameters including, e.g., hops, latency, and energy efficiency policies. In this context, Software Defined Networking (SDN) has emerged as the answer to these needs decoupling control and data planes, using a logically centralized controller able to manage the underlying networking resources. In this paper, we focus on the adoption of Dijkstra's algorithm in SDN environments to support applications deployed in Cloud/Edge and IoT scenarios. Specifically, considering a highly scalable network topology that includes thousands of network devices, in order to reduce the path computation, we propose a revised MapReduce approach of Dijkstra's algorithm. Experiments show that, compared to the sequential implementation, the MapReduce approach drastically reduces the shortest path computation performance when considering a complex Cloud/Edge and IoT network topology including thousands of virtual network devices.

Keywords: SDN · MapReduce · Dijkstra · Cloud computing
Edge computing · Internet of Things

A. Celesti—On behalf of Gruppo Nazionale Per il Calcolo Scientifico (GNCS) - INdAM.

K. Kritikos et al. (Eds.): ESOCC 2018, LNCS 11116, pp. 185–198, 2018.
https://doi.org/10.1007/978-3-319-99819-0_14

1 Introduction

Nowadays, in the era of Internet of Things (IoT), we are observing a proliferation of new cutting-edge pervasive applications. In this panorama, Gartner [1] predicts that there will be 26 billion of IoT devices by 2020 representing an almost 30-fold increase from 900 million in 2009. Despite the rapid advances of IoT technologies, due to hardware limitations, applications deployed on IoT devices (e.g. Single Board Computers (SBCs), mobile phones, tablets, etc.) have to interact with the microservice architecture hosted by the central Cloud Computing [2,3] data centers and, in order to reduce network latency, also by devices distributed in an intermediate layer called Edge Computing [4].

The microservice architecture is a variant of the traditional Service-Oriented Architecture (SOA) that structures an application as a collection of loosely coupled fine-grained services (i.e., microservices) based on lightweight protocols. The decomposition of applications into different smaller services allows to improve modularity, making them simpler and more resilient. Specifically, applications require the interaction of different smaller services or microservices generally spread in the Cloud, Edge and IoT layers over a wide geographical network. This introduces an important issue: ICT operators must flexibly manage the network in order to meet the requirements of today's emerging applications. In fact, network awareness [5] is fundamental during the deployment of microservices on Cloud, Edge and IoT devices. Unfortunately, ICT operators are not able to have a view of the whole network topology and to think about quickly changing the setup of the physical network assets, if needed, in order to meet the requirements of their hybrid Cloud/Edge/IoT applications [6–8], because network connections are generally shared among different providers. Furthermore, this would cause management problems for Internet Service Providers (ISPs). Therefore, ICT operators need an alternative solution that allows them to gain an advanced awareness of Cloud/Edge and IoT networks in terms of flexible packets routing in order to compute paths according to different parameters including, e.g., hops, latency, and energy efficiency policies.

Driving the need for a new networking solution, Network Function Virtualization (NFV) was introduced with the purpose of building networks without being dependent on ISPs. In particular, Software Defined Networking (SDN) has emerged as the answer to these needs by decoupling control and data planes, using a logically centralized controller able to manage the underlying physical resources of the network, abstracting them to allow ICT operators to perform rapid and automatic configuration of network routing. The ability to dynamically define the behavior of a network via SDN gives ICT operators the flexibility to adapt the network to applications' requirements, without complex and expensive reconfiguration tasks on physical network devices.

One of the main algorithms adopted for network routing is Dijkstra that allows to find the shortest path between two nodes. This algorithm has been recently adopted in SDN environments.

In this paper, unlike the scientific works available in the literature, in order to address a Cloud/Edge and IoT scenario that includes a large number of network nodes, we propose a revised MapReduce version of Dijkstra's algorithm to optimize the connections required by applications whose microservices are deployed over Cloud/Edge and IoT environments.

The experiments carried out haves shown that, with a minimal configuration of the Hadoop cluster - 3 computational nodes and an input dataset describing a complex Cloud/Edge and an IoT network topology with 10.000 virtual network devices, since the number of devices present within the network increases the path computation time performed with the MapReduce approach drastically improves up to approximately 92% compared to the the sequential one.

The remainder of the paper is organized as follows. In Sect. 2, we present an overview of related works and contributions. Motivation is discussed in Sect. 3. In Sect. 4, we introduce some preliminary knowledge regarding the SDN concept. Starting with a sequential implementation of Dijkstra's algorithm for SDN environments, a revised MapReduce version is presented in Sect. 5. Section 6 shows the simulation results and observations. Finally, this paper is concluded with Sect. 7.

2 Related Work

Recently, several initiatives have been proposed regarding the application of the Dijkstra's algorithm in SDN. The limits of traditional hierarchical architecture design principles based on Dijkstra's algorithm in the perspective of emerging Cloud/Edge computing systems are highlighted in [9]. One of the major challenges is the mapping of virtual networks onto physical network infrastructures, which is defined as a Virtual Network Embedding (VNE) problem. In this context, a surviving virtual network mapping problem was formulated and solved using an SVE Survivable Heuristic (GRC-SVNE) algorithm based on the Dijkstra's algorithm proposed in [10]. Furthermore, an alternative GRC-M algorithm in combination in combination with the Multicommodity Flow (MCF) algorithm is discussed in [11].

The application of the Dijkstra's algorithm in SDN raises numerous challenges in terms of reliability, capacity control and scalability. The application of network virtualization in Fiber-wireless (FiWi) networks with the purpose to alleviate bandwidth tension when a physical link serving different virtual networks fail is discussed in [12]. In particular, a shared protection mechanism is embedded within the Dijkstra's routing algorithm in order to improve its reliability when a physical link fails. A reliable security-oriented SDN routing mechanism, named RouteGuardian, which considers the capabilities of SDN switch nodes combined with a piece of Network Security Virtualization framework is proposed in [13]. In particular, it overcomes the limits of the traditional routing mechanisms in SDN, based on the Dijkstra's shortest path, in terms of capacity control in order to prevent network congestion. A self-adjusting architecture based on Pairing heap to scale SDN network overcoming scalability issues due to a centralized control plane is proposed in [14]. By using Network Virtualization Function (NVF), the whole network is viewed as a huge heap and divide it into several sub heaps repeatedly until get the basic units of physical switches in the network. In this context, the Dijkstra's algorithm is applied and optimized based on Pairing heap outperforming the original one when the network is dense.

Dijkstra's Algorithm has been recently used in many emerging applications based on SDN. In [15], an autonomous agent based shortest path load balancing using the Dijkstra's algorithm was proposed to find the shortest path to virtual machines when a Cloud services saturates its processing capabilities. A piece of framework to lightweight process the 3D shape based on Web Browser considering Web3D technology areas in the era of "Internet plus" is discussed in [11]. This framework is based on Mesh Segmentation. Therefore, a new Dijkstra-based mesh segmentation approach is presented. The application of SDN/OpenFlow in an Internet Protocol Television (IPTV) multicasting implementation is proposed in [16]. In this context, an important function of IPTV multicasting is the Joint/Leave request of client in a multicast group. In order to obtain an efficient IPTV service routing, Dijkstra's and Prim's algorithms were used to comparatively calculate minimum total edge weight. Moreover, the Mininet environment is used to emulate the system, that consists of Open vSwitch and a POX controller. Experiments compare the transmission time of the first joint/receive packet to a client when using Dijkstra's and Prim's algorithms. In [17], the Service Function Chaining (SFC) was used as a model of the Shortest Path Tour Problem in order to find the minimum transmission cost path by exploiting a constructed multistage graph. In particular, the minimum transmission cost paths for multiple SFC classes is derived using the Dijkstra's Shortest Path Algorithm with resource constraints in a flexible way. Finally, some experiments are carried out and the results show the effectiveness and efficiency of our proposed method.

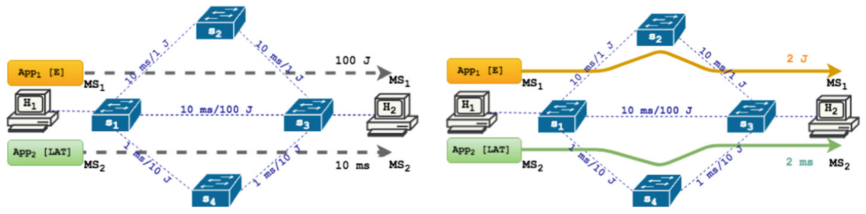
Differently from the scientific work available in literature, in this paper, we focus on a Cloud scenario based on SDN in which the Dijkstra algorithm can benefit from parallel processing in order process a huge amount of virtual network nodes in order to assess best paths.

3 Motivation

With reference to those applications whose structure obeys the microservice architecture in which microservices are deployed on devices across the Cloud, Edge and IoT layers, there is the need to optimize certain network parameters to align applications requirements in terms of latency and energy consumption with network connections.

Figure 1 illustrates a scenario that includes two hosts, H_1 and H_2, that need to communicate through a network topology including switches S_1, S_2, S_3 and S_4. Moreover, we consider two applications App_1 and App_2 that run on both host H_1 and host H_2. App_1 consists of microservice MS_1, while App_2 consists of microservice MS_2.

Suppose App_1 wants to take care of energy consumption, while App_2 wants to take care of latency minimization; applying the shortest path routing approach



(a) Shortest path (b) Latency/Energy aware

Fig. 1. Routing approaches

shown in Fig. 1(a), the connections between H_1 and H_2 pass through the switches S_1 and S_3. This approach is not the best in terms of energy consumption and latency minimization because we obtain $100J$ of energy for App_1 and $10ms$ of latency for App_2.

Figure 1(b) shows an alternative latency/energy aware routing approach that allows to better optimize network resources and paths according to different application requirements. In fact, although they share the same source and destination hosts, App_1 and App_2 are routed separately according to their requirements. In particular, with regard to App_1, the connections between H_1 and H_2 pass through S_1, S_4 and S_3 with $2J$ of energy consumption, while as regards App_2, connections between H_1 and H_2 pass through S_1, S_2 and S_3 with $2ms$ of latency. Furthermore, it is possible to organize customized network connections between H_1 and H_2 for each application such as:

- **Simplex.** Transmission is allowed in only one direction: H_1 always acts as a transmitter, while H_2 acts as receiver.
- **Half Duplex.** Transmission is allowed in both directions, but not simultaneously: when H_1 acts as a transmitter, H_2 acts as a receiver.
- **Full Duplex.** Transmission is allowed in both directions at the same time: both H_1 and H_2 act, at the same time, as transmitter and receiver.

The objective of this paper is to combine shortest path and latency/energy aware routing approaches for SDN environments supporting Cloud, Edge and IoT scenarios. In order to achieve this, we adopt the Dijkstra's algorithm. Although several scientific works have been recently proposed focusing on the adoption of Dijkstra's algorithm in SDN, in this paper, we focus on a revised MapReduce approach of this algorithm that can improve processing times when thousands of Cloud, Edge and IoT devices are considered.

4 SDN Overview

SDN technology is an emerging network architecture in which network control is decoupled from forwarding and directly programmable. The migration of control logic, closely linked to individual network devices, to accessible Cloud, Edge and

IoT devices, allows to abstract the underlying networking infrastructure giving, to applications, a virtual vision of the network. Management is centralized in a purely software SDN controller that has a global view of the network. As a result, the network appears to applications as a single logical switch. With SDN, it is possible to achieve the control of the network, from a single point, regardless of ISPs and network assets, simplifying network design and usage. Moreover, SDN abstraction also simplifies the operation of the network devices, as they no longer need to understand and process thousands of standard protocols, but they simply have to accept instructions from the SDN controller.

Basic SDN operations are performed by a standard protocol that allows the SDN controller to send instructions to the various switches. OpenFlow is one of main open protocols that allows an intermediate communication plane between the SDN controller, i.e., the control plane device, and routers/switches, i.e., data plane devices, that enforces network policies. In particular, OpenFlow routers and/or switches include one or more flow tables and/or group tables updated by an OpenFlow controller that can add or delete flow entries responsively or proactively. Several OpenFlow controller solutions are OpenDaylight, Floodlight, POX, Pyretic, and so on.

Figure 2 shows the general architecture of latency/energy aware applications over OpenFlow. Looking up at the top of Fig. 2, different applications with specific network requirements interact with the OpenFlow controller that monitors their network latency and energy consumption by receiving information from OpenFlow network devices. If a particular network latency and energy consumption parameter does not meet the requirements of an application, the OpenFlow controller enforces network changes to OpenFlow devices.

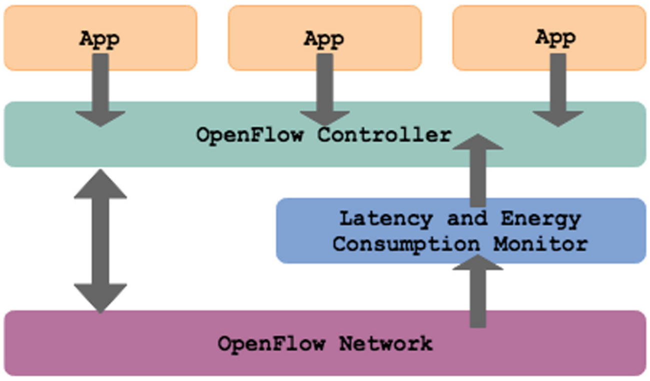


Fig. 2. Architecture of latency/energy aware applications.

5 Dijkstra's Algorithm

Dijkstra's algorithm is very useful in deriving the best routing path for sending packets from a specific source node to a destination node in an SDN environment where different parameters (such as hops, latency and energy consumption) associated to each link in the network must be considered in order to meet the application requirements deployed in Cloud/Edge and IoT scenarios. Suppose we can derive from the SDN topology a graph $G = (V, E)$, which is weighted, directed and connected. Figure 3 shows an example of a real/virtual network through a weighted, directed and connected graph. Therefore, V represent the set of network devices and E the set of network links, while to each link is associated a weight $w[e]$ quantifying different network parameters.

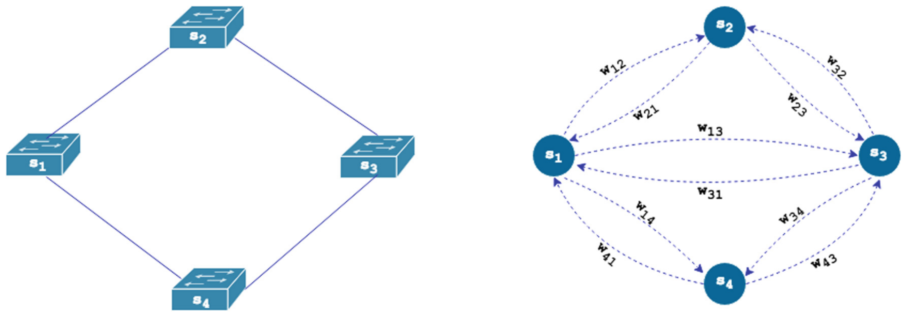


Fig. 3. Representation of an SDN topology through a weighted, directed and connected graph.

A full duplex connection between two network devices can be arranged as a pair of simplex connections each represented as a directed edge. Given that, we suppose there is at least a path between a network device to each other. Considering the latency minimization as an application preference, we assume that $w[e]$ quantify the latency associated to the edge that connects two nodes. A latency-oriented application will be provided with a path with lowest latency between the source and intended destination microservice deployed in the Cloud, Edge or IoT layers. Allocated virtual paths will be periodically updated as underlying physical network changes to ensure a given latency requirement.

Considering that there are many factors that affect the properties of the connections, network changes are more and more varied and unpredictable because the Cloud/Edge and IoT networking scenarios are very complex. This complexity is due to the fact that virtual paths that directly connect two nodes, actually pass through tunnels and/or overlay networks, built on different physical network devices distributed in the Cloud, Edge and IoT layers, that can frequently change.

5.1 Sequential Approach

One of the most common problems in graph theory is represented by the single-source shortest path problem. Moreover, the task deals to find the shortest paths from a source node to all other nodes in the graph. In particular, edges are associated with costs or weights, in which case the task is to compute lowest-cost or lowest-weight paths.

Given a weighted, directed and connected graph $G = (V, E)$, with V the set of vertexes and E the set of edges, the Dijkstra's algorithm uses a Greedy strategy to solve the problem of minimum paths with single source s of the graph $G = (V, E)$ if all the weights are non-negative.

Algorithm 1 shows the sequential Dijkstra's algorithm pseudo-code, whose input is a given connected graph $G = (V, E)$ represented with adjacency lists and $w(u, v) \geq 0$ representing the edge weight from a vertex u to a vertex v, and the single source node s.

Algorithm 1. Dijkstra's Algorithm

```
INPUT $G = (V, E)$, $s$
OUTPUT $d|V|$
  $d[s] \leftarrow \emptyset$
  for all $v \in V - \{S\}$ do
    $d[v] \leftarrow \infty$, for each $v \neq s$, $v \in V$
  end for
  $S \leftarrow \emptyset$
  while $q \neq \emptyset$ do
    $u \leftarrow Extract - Min(Q, d)$
    $S \leftarrow S \cup \{u\}$
    for all $v \in neighbours[u]$ do
      if $d[v] > d[u] + w(u, v)$ then
        $d[v] \leftarrow d[u] + w(u, v)$
      end if
    end for
  end while
  return $d$
```

The algorithm maintains a set S that contains the vertexes whose minimum path weight from the source s has already been determined, i.e., for each vertex $v \in S$ it is worth $d[v] = d(s, v)$. The algorithm repeatedly selects the vertexes $u \in V - S$ with the minimum shortest path estimation, inserts u into S, and releases all the edges outgoing from u. Moreover, a queue with priority Q that contains all the nodes $V - S$ is kept, using the respective values d as key.

5.2 MapReduce Approach

The MapReduce approach of the Dijkstra's algorithm is implemented in Hadoop. From an architectural point of view, there are two types of nodes that control the job execution: one JobTracker and several TaskTrackers. The first acts as master node and coordinates all job executions by scheduling all tasks to different TaskTrakers that act as workers. TaskTrackers perform the assigned tasks and send back to the JobTracker reports on the processing status. If a task fails, the JobTracker reschedules it on another TaskTracker. When a MapReduce job is invoked by an user, the JobTracker divides the job into a set of tasks that are assigned to TaskTrackers to process the job in parallel.

As previously discussed, in the sequential approach the key element is represented by the priority queue Q that keeps a globally-sorted list of vertexes by current distance. This is not possible in MapReduce, as the programming model does not provide a mechanism for exchanging global data. Therefore, we adopted a brute force approach known as parallel breadth-first search. First of all, as a simplification, we assumed that all edges have associated unit weights. This assumption allows us to make the algorithm easier to be understood. The basic idea of the MapReduce Dijkstra's algorithm is that iteratively the distance of all vertexes directly connected to the source vertexes is one; the distance of all vertexes directly connected to those is two; and so on.

Suppose we want to compute the shortest path to vertex n. The shortest path must go through one of the vertexes in M that contains an outgoing edge to n. Therefore, we need to examine all $m \in M$ to find m_s, the vertex with the shortest distance. The shortest distance to n is the distance to $m_s + 1$.

The pseudo-code of the parallel breadth-first search algorithm is provided in the Algorithms 2 and 3. As already assumed for the sequential approach of the Dijkstra's algorithm, we consider a connected, directed graph represented with adjacency lists. Distance to each vertex is directly stored alongside the adjacency list of that vertex, and initialized with distance $d[v]$, $v \in V$ to ∞, except for the source vertex, whose distance to itself is zero. Therefore, in the pseudo-code, n denotes the *nodeid* (i.e., an integer) and N denoted the node's corresponding to the adjacency list. Substantially, the algorithm works by mapping over all vertexes and emitting a key-value pair for each neighbor on the vertex's adjacency list. Therefore, the key will contain the *nodeid* of the neighbor, and the value will be the current distance plus one.

To achieve the implementation of the Dijkstra's algorithm using the MapReduce programming model it has been necessary to implement the *Map*() and *Reduce*() functions as follows.

- **Map()** is invoked in the Mapper task for each available vertex within the graph. The output of the Mapper produces different key-value pairs - a key value pair having as key the source vertex, and as value the adjacent vertexes and another key-value pair where the key is given by the source vertex and the value represents the minimum distance value.

– **Reduce()** for each key vertex all distances are gathered together and the minimum between them is chosen. Gathering of distances is performed by the Hadoop framework while the choice of the minimum distance is implemented by the user. The output of the Reducer produces another key-value pair where the key is represented by the respective selected vertex and the value is the minimum distance.

Parallel breadth-first search is an iterative algorithm, in which each iteration corresponds to a MapReduce job. At the first iteration, the algorithm discovers all vertexes that are connected to the source. At the second iteration, all vertexes connected to those are discovered, and so on. With each iteration, the algorithm expands the search frontier by one hop.

A crucial aspect of the algorithm, is the determination of the number of iterations that it needs in order to finish the computation. Typically, there are *six degrees of separation* suggesting that everyone on the planet is connected to everyone else by at most six steps (the people a person knows are one step away, people that they know are two steps away, etc.). In practical terms, we will iterate our algorithm until there are no more vertex distances that are ∞.

The execution of an iterative MapReduce algorithm requires a non-MapReduce "driver" program, which submits a MapReduce job in order to iterate the algorithm, checks to see if a termination condition has been met, and if not, repeats. The iterative approach is realized using the Hadoop API to construct "counters", which, can be used for counting events that occur during the execution, e.g., number of corrupt records, number of times a certain condition is met, or anything that the programmer desires. Counters can be defined to count the number of vertexes that have distances of ∞: at the end of the job, the final counter value is checked in order to see if another iteration is necessary. The counter values of each worker are periodically propagated to the master. It brings together the values from the completion of the mapping operations and reducing and subsequently returned to the user. The Mapper and Reducer through the use of Reporter can communicate the progress.

Algorithm 2. Mapper Class Pseudo-code

```
Class $MAPPER$
method $MAP(nid\ n, node\ N)$
  $d \leftarrow N.Distance$
  $EMIT(nid\ n, N)$
  for all $nodeid\ m \in N.ADJACENCYLIST$ do
    $EMIT(nid\ m,\ d+1)$
  end for
```

Algorithm 3. Reducer Class Pseudo-code

```
Class $REDUCER$
method $REDUCE(nid\ m, [d_1, d_2, ...])$
  $d \leftarrow \infty$
  $M \leftarrow \emptyset$
  for all $d \in counts\ [d_1, d_2, ...]$ do
    if $ISNODE(d)$ then
      $M \leftarrow d$
    else
      if $d < d_{min}$ then
        $d_{min} \leftarrow d$
      end if
    end if
  end for
  $M.DISTANCE \leftarrow d_{min}$
  $EMIT(nid\ m, node\ M)$
```

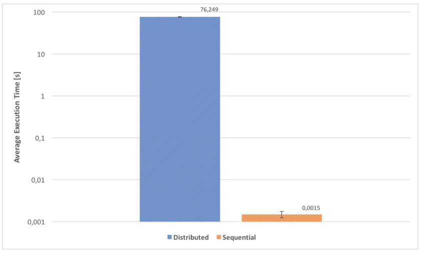
6 Experiments

We carried out a scalability analysis in order to investigate the performance of our sequential and MapReduce implementations of Dijkstra's algorithm. In particular, the scalability analysis is based on the input dataset size to evaluate the average execution time of both implemented approaches. Specifically, we generated several input datasets representing network topologies describing different hybrid Cloud, Edge and IoT scenarios, and of different size (i) 10, (ii) 100, (iii) 1000, and (iv) 10000 vertexes respectively. We remark that in each proposed scenario the vertexes are randomly connected to each other in order to create a weighted, directed and connected graph. In order to have truthful results we performed 30 subsequent iterations of the algorithm for both distributed and sequential approaches and calculated mean values and 95% confidence intervals respectively

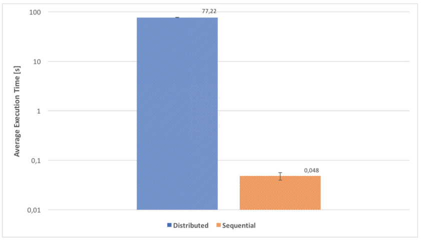
6.1 Experimental Setup

We use three server nodes to deploy the Hadoop MapReduce cluster. Each node has 4 vCPUs at 2.9 GHz, 8 GB of RAM and Ubuntu Server 16.04 LTS, all servers install Apache Hadoop 2.6.1 and JDK version 1.8. The sequential approach of Dijkstra's algorithm, implemented in Java, runs on another server node having the same software and hardware features.

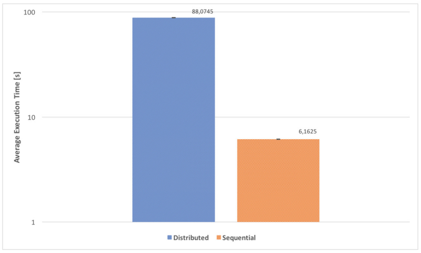
Figure 4(a) illustrates the trend of both distributed and sequential approaches using an input dataset that describes a topology composed of 10 network devices. The execution times of the distributed approach are very large respect to those obtained with the sequential one. This behavior is due to the overhead introduced by the intra-cluster nodes communications. In fact, the MapReduce approach requires roughly 77 s respect to the sequential one which requires only few milliseconds.



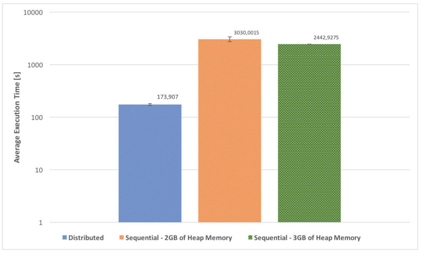
(a) Network topology with 10 devices



(b) Network topology with 100 devices



(c) Network topology with 1000 devices



(d) Network topology with 10000 devices

Fig. 4. Execution Times [s] of the Sequential/MapReduce Approach of Dijkstra's Algorithm (Color figure online)

Figure 4(b) illustrates the mean execution time of both distributed and sequential approaches using an input dataset that describes a topology composed of 100 network devices. The trend is very similar to that obtained in the Fig. 4(a), with the difference that there is a slight variation of execution times. In particular, the execution time of the distributed approach ranges around 79 s, while for the sequential one increases by a couple of milliseconds

Figure 4(c) illustrates the mean execution time of both distributed and sequential approach using an input dataset that describes a topology composed of 1000 network devices. The trend is similar to that illustrated previous two figures. In fact, the execution time registered with the distributed approach increases sligthly by 10 s, the sequential one still be more efficient.

Figure 4(d) illustrates the mean execution time of both distributed and sequential approaches using an input dataset describing a network topology with 10000 network devices. In this configuration, the trend is different. In particular, the execution times collected using the distributed approach are slower - circa 170 s, compared to those obtained through the sequential one.

This behavior is evident also performing a vertical scalability by increasing the heap memory of the JVM from 2 GB to 3 GB. Indeed, with reference to orange and green bars of Fig. 4(d) representing the mean response times of the sequential approach of Dijkstra's algorithm with respectively 2 GB and 3 GB of heap memory reserved for the JVM, the mean processing times are roughly constant and greater than 1000 s. In conclusion, the distributed approach of Dijkstra's algorithm is suitable for huge network topologies (10000 network devices) being 95% and 92% faster than the sequential one configured with $2\,GB$ and $3\,GB$ of heap memory reserved for the JVM.

7 Conclusion and Future Work

In this paper, we considered a scalable SDN scenario, where Cloud, Edge and IoT devices must communicate efficiently to met the application requirements to minimize different network parameters such as hops number, latency, energy consumption, produced CO_2 and so on.

In this complex scenario, a single centralized routing policy can not meet all application requirements at the same time. To achieve this, we considered an SDN environment running a Dijkstra's algorithm to produce routing tables that minimize application network latency.

To address a scalable scenario that includes a huge amount of Cloud, Edge and IoT network devices, in addition to considering a sequential implementation of Dijkstra's algorithm, we also considered a MapReduce implementation to minimize processing times. Specifically, considering small network topologies (up to 1000 network devices), such as that of an intra-building scenario, the sequential Dijkstra's algorithm presents a better mean processing time than the MapReduce one, whereas in a more complex network topology, such as that of an intra-campus or smart cities scenario, in which roughly 10000 network devices are considered, the MapReduce approach represents the optimal solution.

Our future work involves the improvement of our distributed Dijkstra's algorithm implementation to address reliability issues when physical links fail, network capability control, and scalability due to a single control plane.

Acknowledgment. This work has been supported by FP7 Project the Cloud for Europe, grant agreement number FP7-610650.

References

1. Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020. https://www.gartner.com/newsroom/id/2636073
2. Celesti, A., Galletta, A., Carnevale, L., Fazio, M., Lay-Ekuakille, A., Villari, M.: An IoT cloud system for traffic monitoring and vehicular accidents prevention based on mobile sensor data processing. IEEE Sens. J. **18**, 4795–4802 (2018)
3. Galletta, A., Carnevale, L., Celesti, A., Fazio, M., Villari, M.: A cloud-based system for improving retention marketing loyalty programs in industry 4.0: a study on big data storage implications. IEEE Access **6**, 5485–5492 (2017)
4. Ahmed, E., Ahmed, A., Yaqoob, I., Shuja, J., Gani, A., Imran, M., Shoaib, M.: Bringing computation closer toward the user network: is edge computing the solution? IEEE Commun. Mag. **55**, 138–144 (2017)
5. Liotta, A.: The cognitive net is coming. IEEE Spectr. **50**, 26–31 (2013)
6. Celesti, A., Tusa, F., Villari, M., Puliafito, A.: How the dataweb can support cloud federation: service representation and secure data exchange. In: Proceedings - IEEE 2nd Symposium on Network Cloud Computing and Applications, NCCA 2012, pp. 73–79 (2012)

7. Fazio, M., Celesti, A., Marquez, F., Glikson, A., Villari, M.: Exploiting the fiware cloud platform to develop a remote patient monitoring system. In: Proceedings - IEEE Symposium on Computers and Communications, vol. 2016, pp. 264–270 (2016)
8. Mulfari, D., Celesti, A., Villari, M., Puliafito, A.: How cloud computing can support on-demand assistive services. In: W4A 2013 - International Cross-Disciplinary Conference on Web Accessibility (2013)
9. Lin, C., Xue, C., Hu, J., Li, W.Z.: Hierarchical architecture design of computer system. Jisuanji Xuebao/Chin. J. Comput. **40**, 1996–2017 (2017)
10. Zheng, X., Tian, J., Xiao, X., Cui, X., Yu, X.: A heuristic survivable virtual network mapping algorithm. Soft Comput. 1–11 (2018)
11. Zhou, W., Jia, J.: Lightweight Web3D visualization framework using Dijkstra-based mesh segmentation. In: Tian, F., Gatzidis, C., El Rhalibi, A., Tang, W., Charles, F. (eds.) Edutainment 2017. LNCS, vol. 10345, pp. 138–151. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65849-0_15
12. Liu, Z., Yang, H., Kou, S.: Shared Protection Algorithm Based on Virtual Network Embedding Framework In Fiber-wireless Access Network (2017)
13. Wang, M., Liu, J., Mao, J., Cheng, H., Chen, J., Qi, C.: Routeguardian: Constructing. Tsinghua Sci. Technol. **22**, 400–412 (2017)
14. Wang, C., Yan, S.: Scaling SDN Network With Self-adjusting Architecture, pp. 116–120 (2017)
15. Vig, A., Kushwah, R., Tomar, R., Kushwah, S.: Autonomous Agent Based Shortest Path Load Balancing in Cloud, pp. 33–37 (2017)
16. Rattanawadee, P., Ruengsakulrach, N., Saivichit, C.: The Transmission Time Analysis of IPTV Multicast Service in SDN/OpenFlow Environments (2015)
17. Liu, F., Chen, X., An, W., Peng, Y., Cao, J., Zhang, Y.: Minimizing Transmission Cost for Multiple Service Function Chains in SDN/NFV Networks, vol. 2017, pp. 1–6 (2018)