Optimizing Parallel Dataflows in Software-Defined Networks

David Grigorjan, Mohammad Hammad, Robert Oberbeck, Nico Scherer, and Maxim Tschumak

Technical University of Berlin
Electrical Engineering and Computer Science
Complex and Distributed IT-Systems
10623 Berlin, Germany

Abstract. To process large amounts of data in an efficient manner, execution is often scheduled in parallel on multiple nodes. Parallel dataflow frameworks can handle this task but its schedulers currently lack network awareness. Further, dynamic networking, so called *Software-Defined Networking*, is arising. To improve the efficiency in terms of total execution time of data processing, this paper presents an approach of coupling the Apache Flink data processing engine with a Software Defined Network. Different approaches of scheduling tasks to nodes are introduced and discussed. Also, a middleware is developed that handles topology adjusted execution job placement and network-link load balancing, leveraging Software-Defined Networking capabilities, making data processing overall faster and more efficient. Deployment on a software-based testbed shows performance gains from this approach.

Keywords: Software-Defined Networking, Distributed Data Processing

1 Introduction

This paper deals with optimizing performance of parallel dataflows in *Big Data Analytics* utilizing *Software-Defined Networking* (*SDN*). The significance of Big Data Analytics software stems from the prevailing need of many businesses to process large volumes of data. However, many of such systems do not consider the network topology they are deployed on.

Throughput and processing time of an analytical program running on distributed clusters can be improved by utilizing knowledge about the underlying network or managing the network devices dynamically during runtime. We present some approaches and evaluate one of the more promising among them.

We test if, how and by what margin advantages of SDN-based environments over traditional ones increase the performance of a specific data processing engine for use in clusters. We adjust the scheduler of the engine and develop a middleware that communicates with it as well as with our network's control component such that information on the network is delivered to the scheduler on demand.

2 Background

2.1 SDN

An SDN-enabled network provides a *Controller* unit or system that comprises the control logic of the network devices located on separate computer resources resulting in the differentiation between the so called *data-* and *control-plane*. It features a *Southbound API* for conveying information *"down"* to the devices in the data plane as well as a *Northbound API* for communication with application and business logic *"up top"*.

SDN Controllers thus provide a centralized view on the network and could potentially be used to retrieve information for utilization by the data processing platform in deployment decisions. Furthermore, the functions available for network manipulation could be invoked during runtime based on the knowledge about currently running analytical jobs.

2.2 Apache Flink

For this paper, we used Apache Flink as our data processing engine. Flink is a *Fast and Reliable Large-Scale Data Processing engine*. A Flink cluster can be made up from a number of physical computation nodes or even a single host. Processing programs, written in Java or Scala by the user, are automatically adapted by the optimizer and runtime to execute efficient on a given setup.[1] The result of this adaption is the so-called *Execution Graph*, which is deployed. A master-node, the *Job Manager* is responsible for doing so and keeping track of the the overall job execution. The job is split up into several tasks that can be handled independant by *Task Managers* - the actual computation nodes. Depending on the physical hardware, a task manager can offer multiple processing slots. A sketch of the scheduling is depicted in Figure 1 and 2.

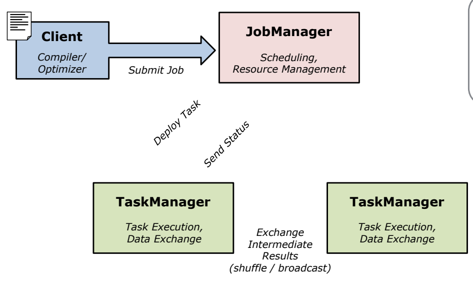
Client
Compiler/ Optimizer
Submit Job
JobManager
Scheduling, Resource Management
Deploy Task
Send Status
TaskManager
Task Execution, Data Exchange
Exchange Intermediate Results (shuffle / broadcast)
TaskManager
Task Execution, Data Exchange

Fig. 1: Sketch of Job Execution[2]

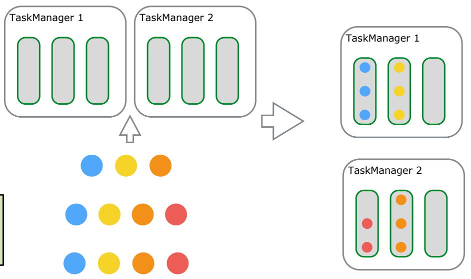

Fig. 2: Execution-Graph embedding to Task Managers[3]

[1] (2015) Apache Flink home. Retrieved March 20, 2015, from https://flink.apache.org

3 Approach

To schedule processing tasks on software-defined networks, we identified two approaches, which we named *Top-Down* and *Bottom-Up*. A sketch of these concepts and their relation to the setup is depicted in Figure 3 and described in the following section.

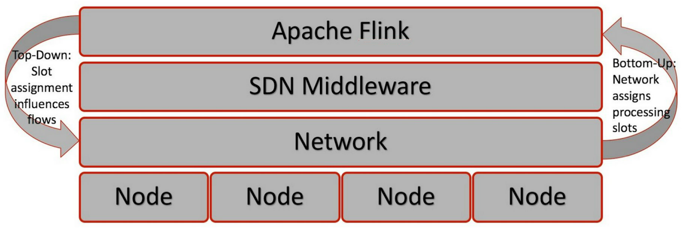
Apache Flink
SDN Middleware
Network
Node
Node
Node
Node
Top-Down: Slot assignment influences flows
Bottom-Up: Network assigns processing slots

Fig. 3: Different scheduling approaches

We make the following assumptions:

- Jobs are not scheduled in parallel but one after another. Otherwise, the scheduler may interfere with scheduling multiple jobs concurrently.
- Jobs always communicate All-To-All. This holds true for the experiments conducted within this paper and eliminates the need to traverse the execution graph. Also, this benefits the generic approach since no deep Flink requirement is built up and the application can be more easily ported.
- Due to the fact that network behaviour is to be studied, the focus lies on network communication. If a single task manager offers multiple processing slots, network effects are weakened. Therefore, a task manager consists of a single processing slot in our setup.

These constraints limit real-world applications but can easily hold true for our test setup to determine performance gains with the concepts introduced.

3.1 Top-Down/Flow Switching

The Top-Down approach imposes that the data processing engine schedules tasks as usual and does not take network properties into account. The middleware is aware of these scheduling decisions and influences the network properly. This influence could be either providing additional bandwidth between communicating

[2] (2015) ClientJmTm.svg Github/Apache/Flink-master from https://raw.githubusercontent.com/apache/flink/master/docs/img/ClientJmTm.svg
[3] (2015) Slots.svg Github/Apache/Flink-master from https://raw.githubusercontent.com/apache/flink/master/docs/img/slots.svg

hosts through flow switching or limiting bandwidth on the specific communication links for background activities. Advantages of this approach are that there is no modification on the scheduler required and it is more portable since only a data processing engine-middleware interface must be implemented. A major disadvantage is the possible worst case: Parts of a job could be scheduled in a data center and another part far away like in another network region.

3.2 Bottom-Up/Scheduling

The Bottom-Up approach required the data processing engines' scheduler actively asks the middleware for a node to place the task on. The middleware is aware of the network topology and can determine optimal task placement through proper algorithms. The disadvantage of this approach is that the data processing engines' scheduler needs to be modified to interact with the middleware. Advantages would be that the middleware is in full control of job placement what not only ensures optimal placement but also can add optional constraints like only use certain network areas for processing during maintenance etc.

A detailed description of the actual implemented algorithms used is given in section 4.4.

4 Implementation

To evaluate the approaches described in the previous section, concrete realizations of an SDN controller and data processing engine is needed. With Apache Flink, the data processing engine constraint was already given. Besides that, a middleware software has to be implemented as a layer between network and high level data engine. To achieve the project goals, a middleware has to be implemented to connect low level network management via SDN controller and high level distribution of the data processing tasks. We implemented the approaches usind *Apache Flink* and *OpenDaylight*.

4.1 SDN Controller

We use the OpenDaylight (ODL) as our SDN Controller. It features good interoperability as well as scalability. It communicates via OpenFlow with the virtual switches for routing and forwarding purposes as well as via a REST-interface with custom programs, acting as a server providing network information on demand.

4.2 Requirements

In the previous sections, two approaches have been introduced: scheduling and flow switching. Implementation of the logic and algorithms relies on some requirements to the middleware software. These are the summarized requirements:

- retrieve information about network topology from SDN controller
- host management (scheduling)
- load balancing (flow switching) through SDN controller
- visualization of current network status

First, the middleware has to obtain information about the network topology and its current status. To get information about the underlying network, a REST client is required to communicate with the Northbound Topology API of OpenDaylight controller. Another requirement to the specification of the middleware is the realization of the scheduling approach. More specifically, it is the implementation of an algorithm of how the physical hosts (slots) are assigned to the tasks of the data processing engine. This should be achieved in a way that the network communication between hosts is efficient and the network resources are utilized as much as possible. The host assignment should be based on the information about the network topology and the current status. The Top-Down approach tries to optimize the network utilization during runtime. The middleware needs to have information about the currently executing job and its deployment (and distribution) in the network. This information is provided by the data processing engine and should be transferred to the middleware. The middleware has to analyze the data and optimize data flows by setting network flows through SDN controller. As an optional requirement, a graphical representation of the current network topology and the host assignment may be implemented. This functionality can be useful for monitoring, controlling and debugging of the middleware and the current job execution.

4.3 Middleware Architecture

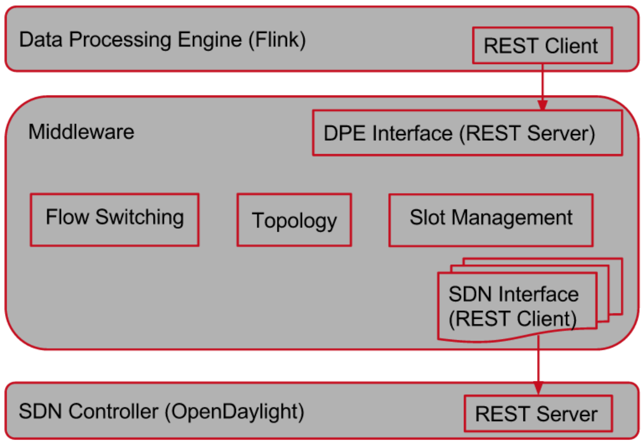
Data Processing Engine (Flink)
REST Client
Middleware
DPE Interface (REST Server)
Flow Switching
Topology
Slot Management
SDN Interface (REST Client)
SDN Controller (OpenDaylight)
REST Server

Fig. 4: Middleware Architecture

The architecture of the implemented middleware software is shown in Figure 4.

The highest level in the architecture is the data processing engine Flink. After the creation of the execution graph and before deployment on the hosts, a communication client has been added to the Flink's scheduler internals to communicate with the middleware. This client sends the execution graph and requests new hosts for the job as soon as they are needed. The hosts, which finished execution are released through a call of this client as well. The corresponding server side is a part of the middleware.

The topology module collects all the data about the network: topology (network devices, hosts and connections between them) and the current utilization of the network. The slot management part is responsible for slot allocation for the distributed data engine. Algorithms behind it and the generic API belong to this module as well.

The flow switching module takes care about the optimization of the network flows based on the current execution graph and network status. In order to access the Northbound API of the SDN controller, generic interfaces were defined so that the middleware does not depend on one particular implementation of a SDN controller. The concrete realization of northbound clients was done separately for the required APIs, such as Flow Programmer and Topology API.

The network visualization module was implemented as well as part of the middleware.

Technically, the middleware is implemented as a Maven multi module project. This decision was made to take advantage of the benefits of a structured project powered by a feature rich build system: flexibility, customizability, dependency management, task automation and more. All the algorithms are covered with tests and the source code is published under Apache License.[4]

4.4 Middleware Slot Assignment

The I/O operations are often the bottleneck of distributed data processing setups. [3] Besides reading and writing to disk, there is the communication layer over the network. The Bottom Up approach should help to organize the execution slots in a way that they are able to communicate in an efficient manner over the network. In this section, the algorithm for slot distribution over the network is described.

As a starting point, there is the execution graph (Figure 5) and the network topology (Figure 6). This information derives from data processing engine and SDN controller respectively. Besides that, it is possible to collect data about the current utilization of the network over the SDN controller. The execution graph represents the steps of the job execution and the network topology is a formation of hosts, network devices and links between them in the network. Both

[4] (2015). Distributed Systems Master Project WS14/15 · GitHub. Retrieved March 12, 2015, from https://github.com/citlab/vs.msc.ws14

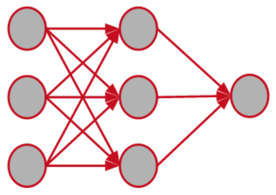
Fig. 5: Execution graph

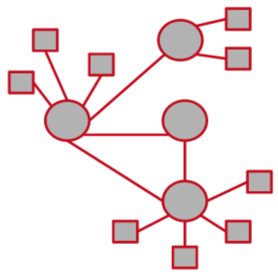
Fig. 6: Network Topology

of them are directed graphs. The question is how to map one graph into another to achieve the best links between the executors.

The general approach is to find an isomorphism between the two graphs. But first, the execution graph has to be normalized since Flink (and other data processing engines) will more likely execute some of the nodes of an execution graph on the same physical host (or even on the same slot). Because of this, the normalization function of the graph depends strongly on the engine being used, so this solution would be not generally applicable. Furthermore, there are no efficient algorithms available resolving the graph isomorphism problem and the complexity of it is even unknown. [5]

Another reasonable approach is the solution to the general heaviest k-subgraph problem. Using the algorithms to calculate the heaviest subgraph it is possible to get k nodes of the graph which connections have the biggest weight. Data processing engine's deployments of the tasks are not static and can be made during runtime. Because of this, there is no information about the total number of hosts (slots) needed for the complete run and the solution to the heaviest k-subgraph problem cannot be applied to the process engines with dynamic deployments. Besides this, the algorithm is NP-hard by reduction from the clique problem and does only work for undirected graphs. [4]

It was not possible to find an optimal solution based on known theoretical approaches. Therefore, a custom approach has been developed and implemented. Intuition for the algorithm is that all hosts which are directly connected to the same switch (general: network device) may have better links than hosts connected through more than one switch. The number of hubs and link quality also affect the transference of the data.

First, we define a *Host Group*. A network device (circle in Figure 7) and all directly connected hosts (rectangles in Figure 7) constitute a Host Group. Thus, network topology consists of a set of Host Groups and each of them consists of one switch and a set of hosts. Furthermore, we define the quality of the link between the hosts by affiliation of the hosts to the same Host Group.

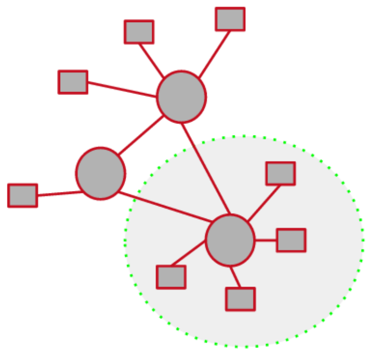

Fig. 7: HostGroup in network topology

$$weight\_full = data\_size/(bandwidth - bandwidth\_used) + latency \quad (1)$$

$$weight\_simple = data\_size/bandwidth \quad (2)$$

If two hosts are not in the same Host Group, the quality of the connection between them is the transmission time for the data needed to be sent between the switches of the two Host Groups where the hosts belong to. This value weight of the edge in the topology graph is used to calculate the shortest paths by the Dijkstra algorithm. In case there are no statistics about current utilization of the network, a default size can be applied.

```
IP: findNextBestHost()
     if no_free_hosts()
          return NULL
     if all_hosts_are_free()
          return get_biggest_host_group().getFreeHost()
     if working_host_group_has_free_hosts()
          return free_host_group.getFreeHost()
     else
          return next_best_host_group().getFreeHost()
end
```

Listing 1: Algorithm of finding free hosts based on Host Group concept

The algorithm in pseudocode is shown in Listing 1. Basically, there are only four options. If there are no free hosts available, the data process engine will not get any host for its calculations. In case that all hosts are free, the biggest Host Group will be selected and the data processing engine receives one of the host for executions. If there is a Host Group with some occupied and idle hosts, one idle host will be returned to the data processing engine. In the last case, there are some Host Groups which are fully occupied. In order to find the next best

Host Group, the algorithm searches for the Host Group with the best link to the Host Groups that is currently occupied using shortest paths calculated by Dijkstras' algorithm.

The implementation of this concept and algorithms is thread safe and optimized to be efficient (caching). Furthermore, the algorithms can be easily adapted to execute different parallel jobs on the same topology at the same time.

5 Outlook

The implementation discussed in this paper focused on scheduling and placement of Flink jobs with knowledge about the physical network. Other issues and enhancements remain open to take complete advantage of network aware job scheduling.

5.1 Traffic Properties

Monitoring would enable better scheduling and flow programming by taking latency and network utilization into account. The OpenFlow specification [7] offers port and flow based packet and byte counters, but neither measures latency, nor bandwidth. Therefore monitoring has to be somehow implemented.

One approach to determine latency between two switches is to insert additional OpenFlow packets and measure the duration. `PACKET OUT` messages forward additional packages to one or more ports. The OpenFlow controller receives a `PACKET IN` message when a switch could not assign a packet to a flow of its flow table. The latency can then be easily calculated by subtracting the time of packet creation and roundtrip times from the time the packet was received via `PACKET IN`. [8] [1]

OpenFlow also supports a `FLOW REMOVED` message that is received by the controller when a flow entry of a switch is expired. This message contains information about the duration an entry was active in the flow table, the amount of traffic matched against an entry and the input port to determine the source of traffic. These information enable the computation of utilization between interswitch links. This approach does not produce any instrumentation overhead since the `FLOW REMOVED` messages are send anyhow. [13]

Another way to measure network utilization is to poll the OpenFlow statistics. The problem is to find an intelligent polling algorithm to reduce the load on switches. Using the routing information of a flow is helpful to find an appropriate querying strategy. A good compromise between accuracy and load balancing is to query two switches randomly and select the one closer to the destination. [9] [1]

There are implementations for the OpenFlow Controller POX [1], NOX [9] and Floodlight [13].

5.2 Load Balancing

Even using the Bottom-Up approach, the possibility to control data flows in the network still remains. The notion of balancing data flows of alike source and sink around multiple paths comes to mind. OpenDaylights' default flow configuration reacts upon activation by a switch clueless to an incoming packets' destination. The controller then prompts the updating of all switches connected to the controller so that future packets to said destination host will follow along the shortest path.

This behavior might be sufficient in the case where sources and sinks are not grouped around common switches but in our case we would have lots of tasks deployed on host groups characterized by single switch connections due to our scheduling decisions. In the assumed case of all-to-all data-flows between two or more host groups, we would not want to send all data along a single path. Even with the switch of each host group posing as a natural bottleneck, we assume better overall load balancing would stem from using more than a single shortest path which we could determine with Yen's Algorithm [12].

6 Evaluation

The Flink Cluster was built on a single physical host using virtualization and the SDN simulator Mininet.

6.1 Mininet

Mininet is a software to simulate SDN environments on local machines. It emulates physical hosts, switches, links and routers on a Linux Kernel.[5]

Since the Mininet-network is emulated in an isolated environment, connected hosts do not have access to the outer world, e.g. the Internet. To establish a connection, a Network Address Translation Router (NAT) has to be connected. This is required to be able to communicate between cluster nodes and the middleware.

To evaluate the performance gain using the approach described in this paper, we need to create a custom topology that resembles currently deployed network infrastructures at data centers as close as possible. A common network structure is the so-called *Fat Tree.* [2] This topology connects hosts and switches in a tree-like manner, with hosts on the lowest layer and switches in the layers above. Since link bandwidth increases with higher layers, the tree is called *fat.* A sketch of this setup is depicted in Figure 8 and 9.

Figure 8 shows a typical Fat Tree setup, Figure 9 the Mininet topology created by us. *sX* represent switches, *hX* hosts with their name (*hX*) and IP-Address (*10.0.0.X*). Link bandwidth is depicted as line width.

[5] (2012). Introduction to Mininet · mininet/mininet Wiki · GitHub. Retrieved March 12, 2015, from https://github.com/mininet/mininet/wiki/IntroductiontoMininet

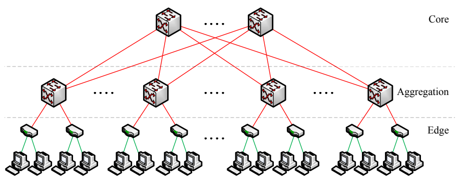

Fig. 8: Typical Fat-Tree setup [2]

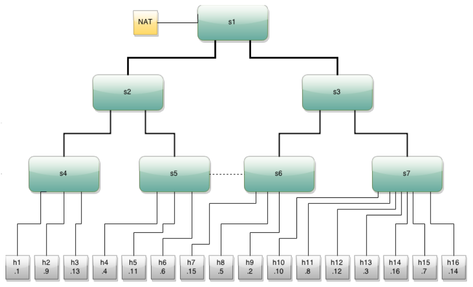

Fig. 9: Created Mininet topology

6.2 System properties

The test setup was built on a single physical host. Its physical properties are listed in table 1.

Property	**Value**
Operating	System Windows 8.1 x64
CPU Intel	Core-i7 3720QM
Memory	8GB 1600MHz DDR3

Table 1: Physical host properties

On top of the physical host, we implemented a virtualized testbed with the properties listed in table 2.

Property	**Value**
Virtualization software	VMWare Player 6.0.4
Operating System	Ubuntu 14.10 x64
Number of CPUs	4
Memory	5100MB
Mininet	version 2.2.0rc2
Open vSwitch	version 2.1.3
OpenDaylight	version 0.2.2-SR2
Flink version	0.7.0 incubating

Table 2: Virtualized guest properties

Within the virtual guest, a mininet environment was built up with the topology described in section 6.1. A Flink cluster was configured to handle every node as worker and use 128MB of heap per taskmanager.

Further, a DoP of 5 was set. For all links, packet loss was set to 0% and delay to 2ms. Bandwidth was set to 5, 10 or 20MBit/s, depending on the link layer.

6.3 Evaluation Results

To evaluate the performance gain of our modified Flink setup, we compared job execution times of the unmodified, native Flink application versus our SDN implementation. For KMeans, we used 5000 datapoints and a variable number of clusters and iterations. The results are shown in table 3.

Number of Clusters	**Iterations**	**Execution time native** $[s]$	**Execution time modified** $[s]$	**Speedup** $[\%]$
5	10	16.823	17.587	-4.54
5	100	38.042	27.367	28.08
50	10	15.195	9.689	36.24
50	100	35.46	19.979	43.66
500	10	13.072	9.166	29.88
500	100	50.547	24.378	51.77

Table 3: Results of evaluation

As one can see, the implemented approach results in an overall speedup of the computation, merely one configuration resulted in a performance decrease. It is remarkable that we gain up to 51.77% performance increase despite the fact that additional roundtrips are required for asking the middleware for hosts to place the task on.

7 Related Work

There have been several efforts of using Software-Defined Networking to increase performance of big data analysis. Application awareness plays a significant role to estimate traffic demand and dependencies. In the case of Hadoop, the Hadoop job tracker offers information about placement of map and reduce tasks, it monitors the progress of all tasks to predict traffic demand for the network-bound shuffle phase. [10] [6] provide ways to program the network at runtime and avoid hotspots by using these information in combination of network topology and link-level utilization.

In [11], Software-Defined Networking is used to optimize distributed analysis queries by prioritizing network traffics and reserving bandwidths. In addition, they match the query plans against the network topology. Topology aware task scheduling is also proposed in [10].

Our work is focused on the placement of Flink tasks by taking network topology into account. The algorithm developed could also be used for other frameworks.

8 Conclusion

Using information about the underlying network, we improved task placement with consideration of physical distances in form of hops, effectively reducing job processing time by eliminating previously existing redundant transmission costs resulting from more arbitrary deployment decisions. More intelligent scheduling via traffic monitoring and the possibility of post-deployment load balancing further encourage the utilization of SDN-enabled networks in conjunction with processing systems for Big Data Analytics.

We evaluated a significant overall performance gain by adding the notion of grouped host deployment. Even though our approach is a general one equivalent solutions in conjunction with alternative software might not be possible due to the open source nature of our setup components. Nevertheless, the efficiency improvement of our setup validates some advantages of its components and should encourage further research of our approach under different circumstances.

References

1. van Adrichem, N.L.M., Doerr, C., Kuipers, F.A.: Opennetmon: Network monitoring in openflow software-defined networks. In: 2014 IEEE Network Operations and Management Symposium, NOMS 2014, Krakow, Poland, May 5-9, 2014. pp. 1–8 (2014)
2. Al-Fares, M., Loukissas, A., Vahdat, A.: A scalable, commodity data center network architecture. SIGCOMM Comput. Commun. Rev. 38(4), 63–74 (Aug 2008)
3. Anderson, D.C., Chase, J.S., Gadde, S., Gallatin, A.J., Yocum, K.G., Feeley, M.J.: Cheating the i/o bottleneck: Network storage with trapeze/myrinet. In: IN PROCEEDINGS OF THE 1998 USENIX TECHNICAL CONFERENCE. pp. 143–154 (1998)
4. Billionnet, A.: Different formulations for solving the heaviest k-subgraph problem. Tech. rep. (2002)
5. Fortin, S.: The graph isomorphism problem. Tech. rep. (1996)
6. Neves, M.V., Rose, C.A.F.D., Katrinis, K., Franke, H.: Pythia: Faster big data in motion through predictive software-defined network optimization at runtime. In: Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium. pp. 82–90. IPDPS '14, IEEE Computer Society, Washington, DC, USA (2014)
7. Open Network Foundation: OpenFlow Switch Specification, version 1.0.0 edn. (2009)
8. Phemius, K., Bouet, M.: Monitoring latency with openflow. 2013 9th International Conference on Network and Service Management (CNSM) pp. 122–125 (2013)
9. Tootoonchian, A., Ghobadi, M., Ganjali, Y.: Opentm: Traffic matrix estimator for openflow networks. In: Passive and Active Measurement. vol. 6032, pp. 201–210. Springer Berlin Heidelberg (2010)
10. Wang, G., Ng, T.E., Shaikh, A.: Programming your network at run-time for big data applications. In: Proceedings of the First Workshop on Hot Topics in Software Defined Networks. pp. 103–108. HotSDN '12, ACM, New York, NY, USA (2012)

11. Xiong, P., Hacigumus, H., Naughton, J.F.: A software-defined networking based approach for performance management of analytical queries on distributed data stores. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. pp. 955–966. SIGMOD '14, ACM, New York, NY, USA (2014)
12. Yen, J.Y.: Finding the k shortest loopless paths in a network. Management Science 17(11), 712–716 (Jul 1971)
13. Yu, C., Lumezanu, C., Zhang, Y., Singh, V., Jiang, G., Madhyastha, H.V.: Flowsense: Monitoring network utilization with zero measurement cost. In: Passive and Active Measurement. vol. 7799, pp. 31–41. Springer Berlin Heidelberg (2013)