SparkL

Large-scale distributed data analytics for weakly-connected clusters

Kristina Shia
Yale University
kristina.shia@yale.edu

Anushree Agrawal
Yale University
anushree.agrawal@yale.edu

ABSTRACT

Systems for large-scale distributed data analytics have been studied extensively and deployed in many commercial production environments. However, most of these systems are deployed within data centers with strongly-connected networks and large bisection bandwidths, but emerging applications require more flexible scheduling methods to optimize overall system performance in spite of network constraints. For example, military use cases may require data and computation to occur thousands of miles apart, as data is generated from sensors in the field and computation occurs in data centers. At the same time, though these sensors are weakly connected to the data-centers, the data analytics may still need to take place in real-time for immediate decision making. After examining existing systems including Apache Spark and Hadoop MapReduce, we designed a novel orchestration framework to optimize these data analytics capabilities in weakly-connected networks.

We present SparkL, a system that uses network topology and bandwidth constraints in a software-defined network (SDN) to optimally schedule data analytics jobs in weakly-connected clusters. Our system utilizes linear programming to calculate job assignments, taking into account bandwidth sharing across the network. We implemented a system consisting of a scheduler, an optimization engine, and computation and storage nodes. Our computation and storage nodes exist within an SDN, controlled and monitored by Unicorn and OpenFlow. In our experiments, SparkL can determine optimal bandwidth allocation and job assignments, which leads to decreases in overall job completion time. SparkL has the potential to be extended for use within Spark and other big data frameworks to orchestrate data analytics in large-scale, real-world applications.

ACM Reference Format:
Kristina Shia and Anushree Agrawal. 2018. SparkL: Large-scale distributed data analytics for weakly-connected clusters.

1 INTRODUCTION

1.1 Problem

In traditional large-scale distributed data analytics, all nodes on which computation occur and data are stored are assumed to be strongly-connected. That is, in the typical data center where these large-scale analytics are run, jobs are not limited by network constraints, and bisection bandwidths are assumed to be very high. This is because the paradigm of many of these data analytics systems is to bring analytics code to the data itself, so the analysis and data storage happen on the same node, or at least within the same data center.

, ,
2018.

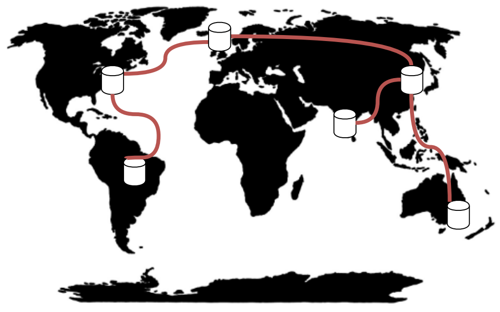

Figure 1: An example weakly-connected cluster

However, there also exist cases where data analytics is necessary for weakly-connected clusters. For example, military settings may necessitate data generation in the field, through sensors, which is then communicated back to data centers and processed in real-time. These data centers may be located miles away, and in this case, bringing analytics to the sensors themselves is not feasible. An example of this situation is depicted in Figure 1. Here, data may exist in nodes, such as sensors, that are not well-connected to computation nodes. In this setting, network constraints may dramatically increase the amount of time data analytics jobs take to run because data download time dominates the overall job time. For example, Spark, a popular distributed data processing engine and the focus of our project, requires data to be fetched and placed on the server where the Spark job will be run, either in memory or on disk if the data is too large. In this weakly-connected setting, transferring data from a military setting or data storage unit in Europe to a computation server in Australia could take much more time than transferring data to the computation node in the United States based on the distance. However, if the bandwidth between Europe and America is much lower than the bandwidth from Europe to Australia, choosing Australia as the computation node would result in quicker download times and a shorter overall job time.

In a distributed job that is broken down into stages, next stages require the output of previous stages. If one poor choice is made with regards to choosing a data source, the task that relies on that data could end up becoming a straggler, where data fetching causes the task to take much longer than other tasks in the same stage. Because the next stage requires all of the output of the previous stage, the entire job is limited by the straggler time.

With current data analytics frameworks, network information is not taken into account, and choosing servers to receive data from is essentially random, or based on limited data locality rules that only help in the case of a strong-connected data center use case.

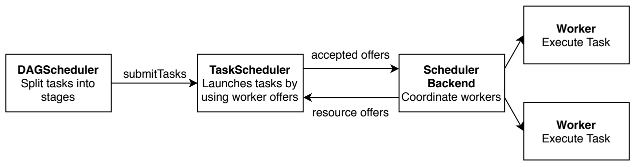

Figure 2: How Spark breaks jobs into tasks and schedules tasks across workers

In this project, we use network bandwidth and connectedness information to create a system, called SparkL, that improves scheduling for distributed data analytics. We use Spark as the core data processing engine for this project to illustrate our system's effectiveness.

1.2 Spark Scheduling

Spark is a multi-purpose data analytics framework that is often used in cluster computing [Zaharia et al. 2010] Spark pioneered the notion of an RDD, or a resilient distributed dataset to deal with intermediate data outputs of computation stages, which exists across many machines and can be rebuilt from previous a RDD if it is corrupted or lost. Spark is used often for machine learning jobs because of their iterative nature and has been shown to be particularly efficient due to its in-memory data processing.

Jobs in Spark are broken down into stages, which are further broken down into tasks. Stages use RDDs from previous stages to build upon computation. Spark has a class called `DAGScheduler` that is responsible for splitting the overall graph of tasks into a set of stages, and the `DAGScheduler` then submits these stages to a `TaskScheduler`. The tasks in a particular stage are not dependent on one another so they can be executed in any order and on any set of nodes. A `TaskScheduler` uses either the First-In-First-Out (FIFO) or FAIR method to assign tasks to specific workers. FAIR scheduling assigns tasks in round-robin order.

`TaskScheduler` receives "offers" from a scheduler backend. Offers are workers that are available to do work. The scheduler backend manages the workers that are available in a cluster. `TaskScheduler` then assigns tasks to these workers and notifies the scheduler backend. Figure 2 shows more details on Spark scheduling.

1.3 Software-defined networks

Software-defined networks (SDNs) allow for refined management and configuration of network performance by separating the data and control planes. The control plane is implemented with a network controller that maintains a global system view and enforces policies accordingly. In our project, we use the OpenFlow protocol via its integration with Unicorn. Since our setting utilizes an SDN, we can capture a global perspective of bandwidth utilization and capacities.

2 RELATED WORK

2.1 HDFS

The Hadoop Distributed File System (HDFS) is used to store and retrieve large amounts of data in a distributed setting, often used for data analytics work [Shvachko et al. 2010]. HDFS provides abstractions for user applications that want to use data–the data itself is broken down into blocks and distributed across many nodes, but user applications do not have to be aware of this fact. Users only query a `NameNode`, which is a master server that figures out which servers the data blocks should be retrieved from and then tells users where to fetch data from [Apache Hadoop]. Figure 3 shows the HDFS architecture for reading and writing in more detail.

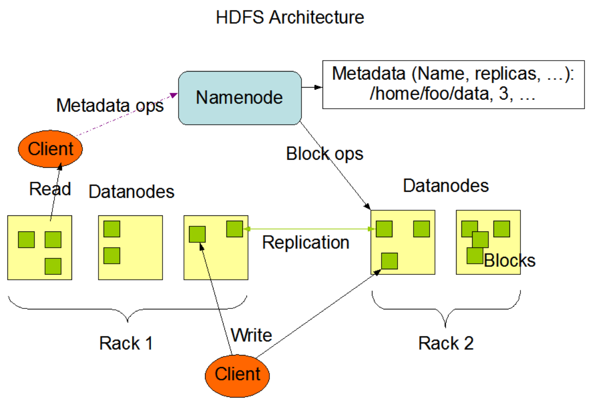

Figure 3: Hadoop Distributed File System architecture. Taken from [Apache Hadoop].

HDFS also relies on replication to be fault-tolerant, so it stores blocks on multiple data nodes. HDFS is often used to serve data to large scale data computation frameworks, like MapReduce. HDFS is not aware of network bandwidth limitations, so while it does divide data across multiple nodes and stores duplicates of these blocks, it does not use network limitations to choose which data node to serve a user's request for data. Data nodes are chosen based on proximity to client, which may miss important bandwidth sharing and constraint information.

2.2 Other Scheduling Modifications

Various other Spark scheduling modifications have been made. On the computation side, Sparrow, which uses a randomized sample approach, was shown to be close to ideal and useful for short, highly parallel tasks that need to be scheduled extremely quickly [Ousterhout et al. 2013].

On the network side, more generally, Sincronia uses the coflow abstraction to order coflows for data center network design. Sincronia does not use a master scheduler, and instead allows each host to use a greedy approach to order coflows, which allows it to come within 4x of the optimal completion time [Agarwal et al. 2018].

Similar in motivation to SparkL, Orchestra is designed to optimize data transfers specifically for frameworks like Spark [Chowdhury et al. 2011]. Orchestra focuses on shuffle and broadcast steps, which are the most traffic intensive because they pass data around different computation nodes in between stages. Orchestra creates a Transfer Controller that behaves similar to BitTorrent for broadcasting data. Orchestra is implemented at the application layer because it directly modifies shuffle and broadcast MapReduce actions, which means it can only be used in the MapReduce context for data analytics.

Additional job scheduling modifications have been made in similar systems like MapReduce model of Hadoop. These replace the FIFO or FAIR scheduling algorithms and incorporate considerations of additional system state. There are several proposed modifications. A Longest Approximate Time to End (LATE) scheduler attempts to reduce the time taken by the slowest running task. A Resource Aware scheduler takes into account the fact that different tasks may have different CPU, memory, and disk requirements while different nodes may have different CPU, memory, and disk resources. As a result, rather than assuming fixed resources for nodes, the executors advertise the number of free computation "slots" they have, and the tasks are matched according to their resource demands [Senthilkumar, M. and Ilango, P. 2016]. Both of these approaches are particularly informative because they attempt to account for heterogeneity in the executor clusters. However, none of these approaches provide a solution for weakly-connected clusters where network bandwidth sharing may be the cause of the bottleneck.

3 METHODOLOGY

3.1 Unicorn

Unicorn is a system built for resource orchestration in "multi-domain, geo-distributed" settings. In this project, we focus on single-domain cases, but take advantage of Unicorn's ability to calculate optimal resource allocations. [Xiang et al. 2017] In particular, SparkL implements an algorithm for computing optimal bandwidth allocation based heavily on the global resource orchestration algorithm presented in the Unicorn paper. We utilize the Unicorn server for resource representation. By querying the Unicorn server, we can obtain the network constraints for a particular set of flows. This becomes part of the input to our optimal bandwidth allocation calculation. Rather than make decisions based on a local understanding of bandwidth limits, SparkL utilizes Unicorn's resource representation to make optimal decisions based on a global understanding of bandwidth capacity, which accounts for limitations for sets of flows due to link sharing.

3.2 Spark Cluster

Spark can be run in multiple configurations: local mode, where the job is run on the local machine without a resource manager, or cluster mode, where a resource manager coordinates execution between different hosts. Spark supports a few different cluster frameworks, including standalone, Mesos, and Yarn. In cluster mode, generally one node is dedicated as the master, which allocates resources and is aware of all of the worker nodes. Worker nodes offer the master node resources.

For our application, we are simulating Spark standalone cluster mode by running multiple Spark instances of local mode on various computation nodes, and then creating our own master node to act as a network resource-aware scheduler between these local instances. In our example, each of our computation nodes could be considered a cluster of its own, weakly connected to other clusters via the network. The key is that we cannot send another stage of tasks until the first set of tasks has finished, which means that if there is a straggler (i.e., a task that takes much longer than the rest), then the time for the entire stage of tasks is bounded by that time.

We chose this method of simulation instead of directly modifying the Spark DAGScheduler for a few reasons. Most importantly, Spark's current scheduling requires data sources for all nodes to be local to the machine, or requires HDFS as a data manager. Because HDFS does not allow direct access to different data sources, changing the DAGScheduler to use different storage nodes would require a large refactor of HDFS. This would make our system too tightly coupled to HDFS and the DAGScheduler. Instead, by simulating the master scheduler, we are able to decouple some of the functionality and make the system more extensible for other data analytics systems, as evidenced in the architecture of SparkL.

3.3 Overall Architecture

Our overall architecture for SparkL (pronounced "sparkle") is show in Figure 4. To demonstrate our system, we used 4 unique server types; 1. Scheduler Server, to act like the "master" in a normal Spark cluster scenario and coordinate scheduling across nodes. 2. CPLEX Server for our linear programming solver, which serves as an optimization engine. 3. Unicorn Server, to discover network bandwidths and also run OpenFlow, our SDN controller. 4. Computation Node Server, which we run on each computation and storage node in our network, to allow us to run jobs and return data files. We created lightweight Flask servers for Scheduler, CPLEX, and Computation Node Servers. Our Unicorn Server and Computation Node Servers run within a Containernet, which is a fork of Mininet that allows hosts in SDNs to run within separate Docker containers. This allows us to control the network topology and bandwidth between computation and storage nodes.

We ran SparkL on two separate Google Cloud Platform instances. One of our instances was reserved for CPLEX Server, and the other instance held our Scheduler Server and the Containernet. We will now discuss the function of each part of the architecture in more detail.

3.4 CPLEX Server

CPLEX is a linear programming solver written by IBM. We created a CPLEX program to properly optimize for the bandwidth and flows that our network should use to communicate from computation to storage node. The endpoints for the CPLEX server are agnostic to the actual linear programming solver and program used for optimization, so the optimization engine does not necessarily need to utilize CPLEX specifically.

3.4.1 Algorithm.

Our algorithm below is adapted from [Xiang 2017]. We utilize two layers of optimization. The first layer calculates the optimal set of flows that minimizes the longest data accessing delay t_j. A second layer is required to maximize the bandwidth utilization. The first layer's objective function only optimizes for selecting the best flows to reduce the time taken by the longest running job. Using the optimal flow set as input, we can then minimize the sum of t_j for all jobs to calculate the maximum bandwidths we can allocate to reduce the overall job time.

The relationship between the flows is captured with the allowed flows matrix L. The allowed flows matrix is created to only allow each computation node and storage node to be selected once in the set of optimal flows. We enforce this constraint to reduce the number of idle computation and storage nodes, but in the case

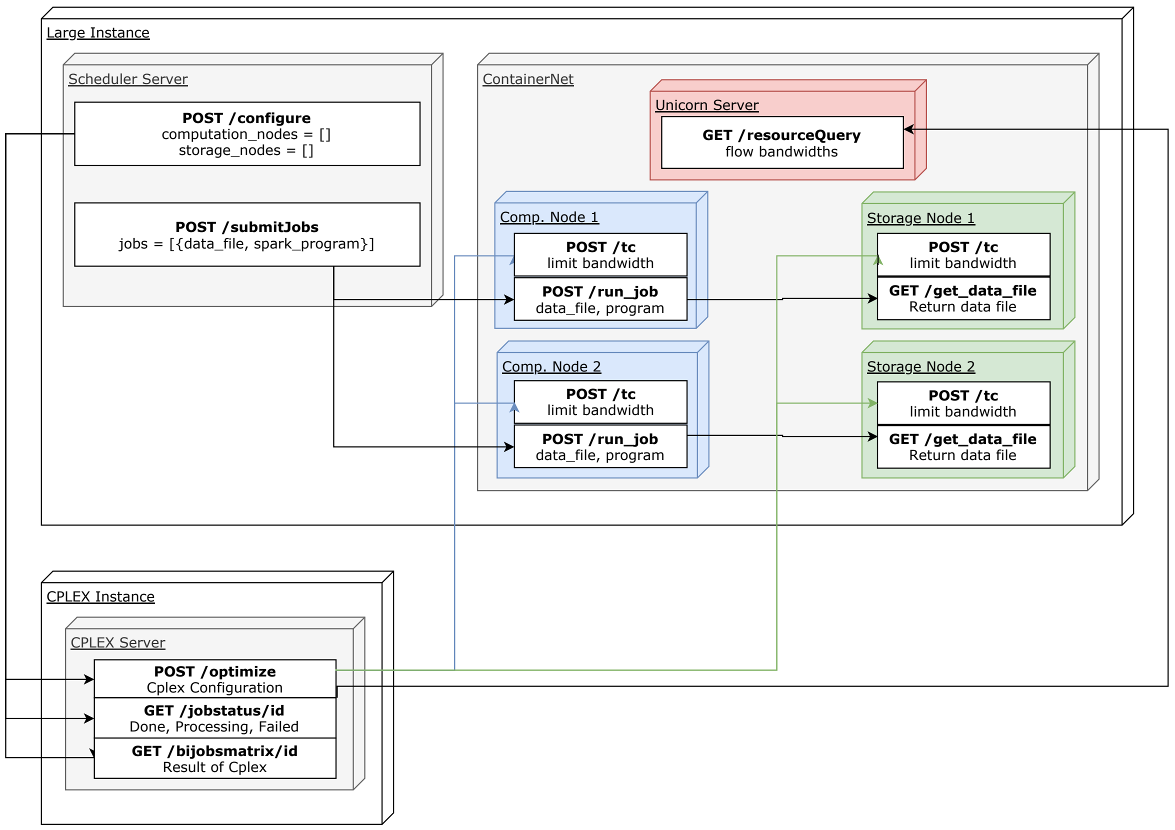

Figure 4: Architecture of our end-to-end system

where the computation or storage resources of some nodes may far exceed those of other nodes, it may be reasonable to loosen this constraint. Allowed flows are represented by 1, while forbidden flows are represented by 0.

The L matrix can be defined as follows:

$$L_{x,y} = \begin{cases} 1 & x_{comp} \neq y_{comp} \text{ and } x_{stor} \neq y_{stor} \\ 0 & \text{otherwise} \end{cases}$$

Let C be the set of all available computation nodes and S be the set of all storage nodes. In the formulation below, J represents the list of all jobs to be assigned. We expect that $|J| \leq \min(|C|, |S|)$. Any additional jobs can be moved into the next job set if no computation and storage nodes are available.

In the constraints, F represents the set of all flows, which is equivalent to all pairwise matching of computation and storage nodes, so $|F| = |C| * |S|$. B represents the optimal bandwidths for each flow. Each I_f^j in the I matrix is a binary variable indicating of a job j has been assigned to flow f.

Below is the first layer of the optimization. From this, we output the I matrix, which is used in the second layer of optimization.

Minimize:

$$max_{j \in J}\{t_j\} \tag{1}$$

Subject to:

$$\sum_{f \in F} I_f^j = 1, \qquad \forall j \in J \tag{2}$$

$$\sum_{j \in J} I_x^j * \sum_{j \in J} I_y^j \leq L_{x,y}, \qquad \forall I_x \in I \text{ and } \forall I_y \in I \tag{3}$$

$$\frac{1}{\sum_{f \in F} B_f * I_f^j} = t_j, \qquad \forall j \in J \tag{4}$$

$$A(BI) \leq C \tag{5}$$

Equation 2 guarantees that for all jobs, exactly one flow is selected. If a flow f has been assigned to any job then, $\sum_{j \in J} I_f^j = 1$. Equation 3 states that any two flows that have both been assigned to jobs must be a pair of allowed flows, meaning they do not share the same computation or storage nodes. Equation 4 calculates t_j, which represents the relative expected data accessing time. Since we currently do not have information about the data size, we use the numerator 1. However, if we had information about the volume of this job, we could substitute that value for the numerator to achieve a better estimate of the job duration.

We received the A and C matrices from the Unicorn server, which are the constraints on bandwidth for our network topology. The A matrix represents any bandwidth sharing of flows and C represents the bandwidth capacities.

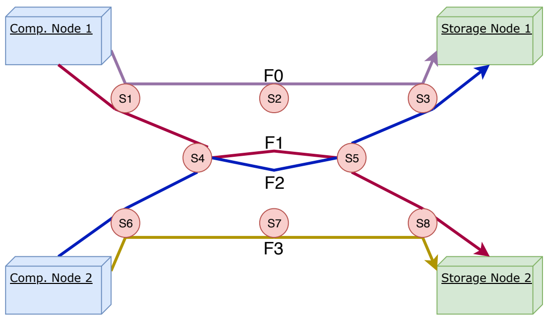

Figure 5: Diagram of the flows we could choose from.

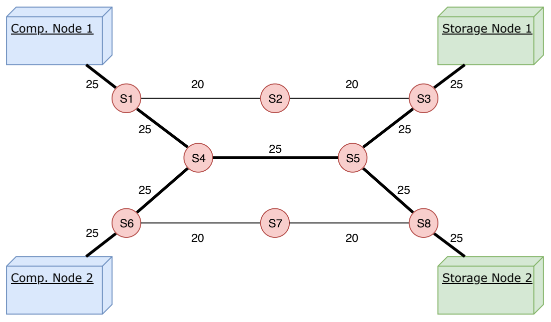

Figure 6: Topology A; bandwidth in Mb

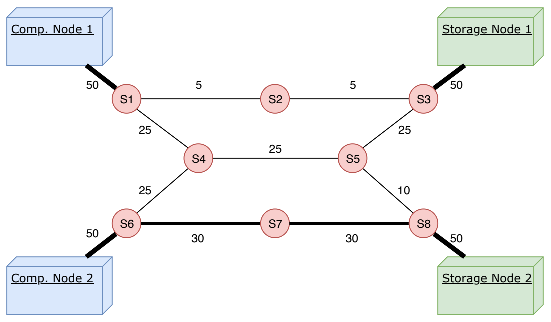

Figure 7: Topology B; bandwidth in Mb

From these constraints, we can calculate an optimal set of flow assignments for all of the jobs, represented by I. Using this optimal set of assignments, we can compute the optimal bandwidths for each of the selected flows in the second layer of our optimization. The model is subject to the same constraints as in the first layer, but the objective function uses a sum to reduce the overall data accessing time (not only the bottleneck):

Minimize:

$$sum_{j \in J}\{t_j\} \tag{6}$$

From this second layer, we can use the BI matrix, computed based on the bandwidths in B and the selected flows in I, as output, which contains the optimal bandwidths for only the selected flows. This output of the CPLEX optimization the captures the flows to choose in our topology, and the bandwidths we should allocate to each flow. We used the `tc` Linux traffic control program on computation and storage nodes to limit the bandwidth as prescribed by CPLEX.

3.5 Containernet

Containernet is an extension of the open source project, Mininet, that allows us to set up a network of Docker containers. Each of these Docker containers is deployed with our custom Docker image and serves as a computation or storage node. Then, we can add switches and links to connect the containers with a custom network topology. We also set the bandwidth capacities for each link, which is enforced with `tc`. Finally, Containernet also allows us to use a custom controller, so we use Unicorn, which includes OpenFlow, as the remote controller for our network.

3.5.1 Network Topology. SparkL aims to optimize scheduling of data analytics for weakly-connected clusters. The topologies that we used to test our system represent a simple representation of this scenario. Using Containernet, we set up 4 nodes, 2 of which serve as computation nodes and 2 of which serve as storage nodes. These nodes are connected links that route through 8 switches. The topology is designed such that routes chosen via a shortest path algorithm will result in flows F0, F1, F2, and F3 [Figure 5]. If Flow 1 and Flow 2 are simultaneously used, they have a shared link between switches S4 and S5, which will result in sharing of bandwidth.

By modifying the bandwidth capacity on specific links, we can simulate a weak connection between two clusters. Due to the shared link, we can observe the effects of selecting optimal or suboptimal sets of flows.

3.5.2 Setup. We've written a number of scripts to setup and teardown the network and containers. To begin, we start the Unicorn server in a Docker container, which serves as our network controller. Then, we start all of our nodes and set up the switches and links in the network. Once this Containernet is running, we register the ports with Unicorn so Unicorn can monitor the bandwidth usage. Then, we deploy the routes that correspond to the four flows illustrated in Figure 5. We also need to inform Unicorn of the bandwidth capacities that we've set in the Containernet. Finally, we can start the servers on every node and copy the sample data files to the storage nodes. Once this setup is complete, we can retrieve bandwidth information as well as bandwidth limitations by flows from Unicorn endpoints.

We used Topology A [Figure 6] and Topology B [Figure 7], with the listed bandwidths, to illustrate the functionalities of our system.

3.6 Computation Node/Storage Node Server

On each of the computation nodes and storage nodes, we ran a simple server that allowed us to request and serve data files, and run and time Spark jobs. We also used this server to call out to `tc` to limit the bandwidths on each of the nodes.

3.7 Scheduler Server

Scheduler Server is the master node for our cluster. The server calls out to the Unicorn server to receive network topology information.

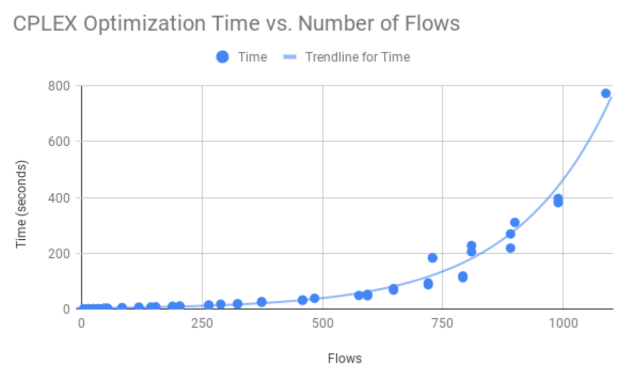

Figure 8: CPLEX Optimization Time for various flows; flows consisted of 2-33 computation nodes and 2-33 storage nodes.

It uses this information to make a request to the CPLEX server to receive the optimal flows and bandwidths to choose. Then, Scheduler Server calls out to the computation and storage nodes to limit their bandwidths. Once configuration is complete, the network can be used with multiple jobs. Scheduler Server handles assigning the jobs to the appropriate computation nodes and receiving results.

4 RESULTS

4.1 Setup

To test SparkL, we followed the setup described in the Containernet section. We also ran the CPLEX Server and the Scheduler Server. We used two topologies, A and B, with which we then configured our system. We evaluated CPLEX Server individually, the entire configuration step, and then the end-to-end time for our configured system.

4.1.1 Spark Program. We used a k-means clustering Spark job written in Python to run as our example job for all of our testing. We chose this because we wanted a job that would require data and use the RDD capabilities of Spark, and a job that would take longer with more data. We did not want a job that, when given large amounts of data, would take exponentially longer than data download time, because that would not be illustrative of cases where the network bandwidth is the constraint.

4.2 CPLEX Performance

We evaluated the performance of our CPLEX server by calculating the time our optimization took for various number of flows, which was dependent on the number of computation and storage nodes. Results are shown in Figure 8.

These results are as expected. With the topologies we modeled with only 4 flows, CPLEX is relatively quick, taking only about a second. However, as the number of flows increase, time increases exponentially. The virtual instance that we used to run the CPLEX server was quite lightweight; to combat this issue, in an actual deployment, a server with more memory and CPU could help. Different linear programming solvers could also be experimented with. Approaches to avoid having the optimization step constrain the

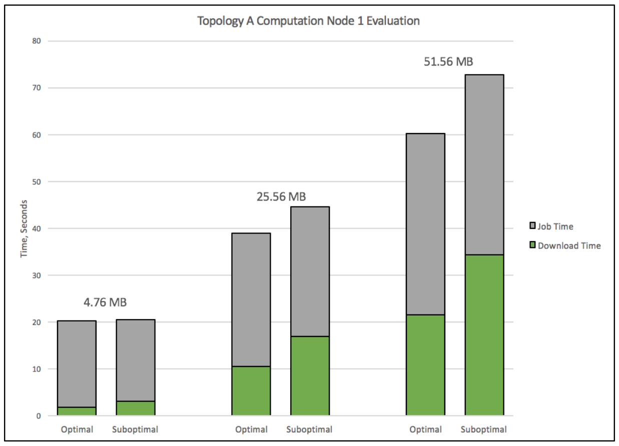

Figure 9: Computation node 1 topology A time results for optimal and suboptimal flows with varying file sizes

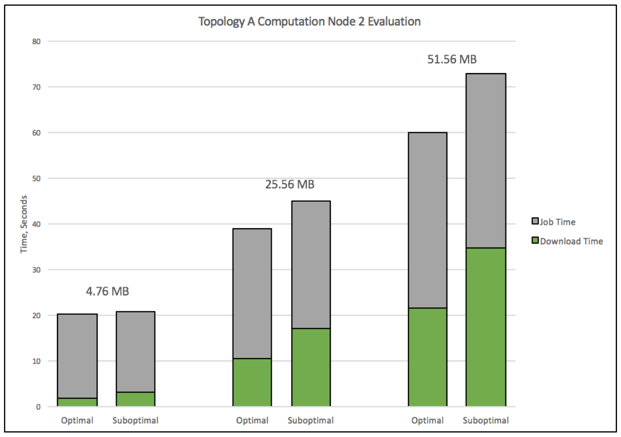

Figure 10: Computation node 2 topology A time results for optimal and suboptimal flows with varying file sizes

performance of the system are proposed in the Discussion section below.

4.3 Configuration Performance

We tested the entire configuration process, which includes receiving network data from the Unicorn Server, CPLEX optimization, and `tc` bandwidth setting. We found that configuring the system with our two topologies with 4 flows, the average configuration time was around 2.54 seconds.

These results show that starting `tc` on the computation and storage nodes and receiving Unicorn data took an additional 1.5 seconds on top of the CPLEX optimization.

4.4 End-to-end Performance

We evaluated the performance of our end-to-end system with two separate topologies (Topology A and B). We timed the two phases of running a job: downloading the data file from the storage node, and running the Spark job with the data. We compared the overall time

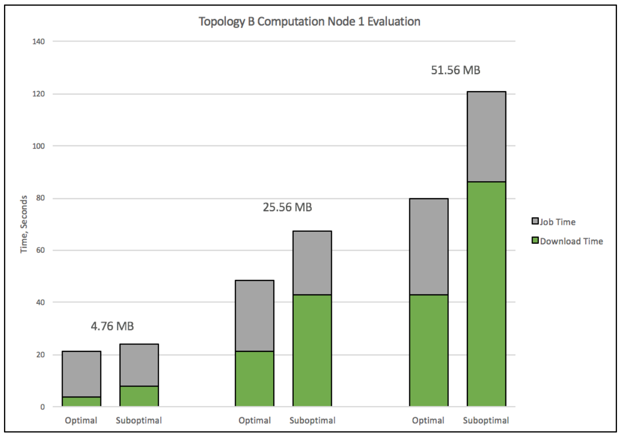

Figure 11: Computation node 1 topology B time results for optimal and suboptimal flows with varying file sizes

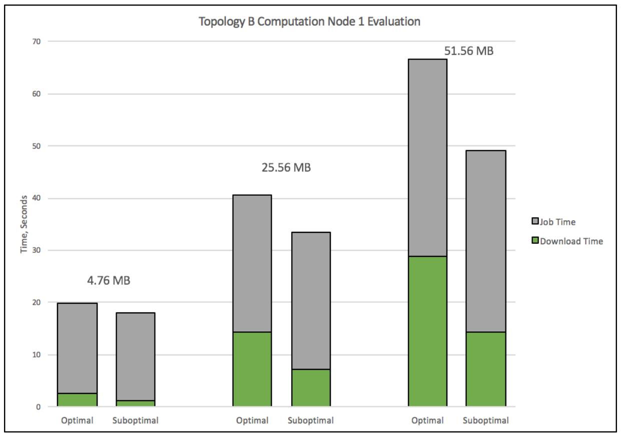

Figure 12: Computation node 2 topology B time results for optimal and suboptimal flows with varying file sizes

and download time per computation node with varying file sizes, and we compared the times for download using the optimal flow calculated by our linear programming solver to the suboptimal flow, which was the exact opposite of the flows chosen by our CPLEX optimization program.

We predicted that independent of which flow was chosen, for a particular size of data, the Spark job would take the same amount of time, since CPU and memory was constant across computation nodes. We also predicted that choosing our optimal flow would result in an overall lower time per stage with jobs j_1 and j_2, which is represented by $max(T(j_1), T(j_2))$ Our results are in Figures 9, 10, 11, and 12, which depict the averages of 4 different runs per situation.

4.4.1 Analysis. With topology A [Figure 6], our optimizer chose Flow 0 and Flow 3, with bandwidths for computation node 1 and 2 as 20Mb. The suboptimal solution was Flow 1 and Flow 2, where the calculated bandwidth for computation node 1 was 12.582Mb and for computation node 2 was 12.418Mb. This is because the suboptimal solution uses the shared link from S4 to S5. We see that as expected, the optimal solution has a shorter download time and overall completion time for the Spark job for both computation node 1 and 2 across all file sizes.

Topology B [Figure 7] was slightly more complicated. Our optimal solution was Flow 1 and Flow 2, with bandwidths for computation node 1 and 2 as 9.998Mb and 15.002Mb respectively. This is because in Topology B, choosing the suboptimal Flow 0 and Flow 3 solution results in computation node 1 receiving a bandwidth of 5Mb, and computation node 2 receiving a bandwidth of 30Mb, and not utilizing the shared link.

If we had a system that only sought to maximize total bandwidth, Flow 0 and Flow 3 would have been chosen since they allow for 35 Mb of bandwidth. The results show that for computation node 2, the suboptimal solution, which uses Flow 3, actually results in faster data transfer on this particular node, which is expected. However, because the stage of the job has to wait for both tasks to finish, the finishing time is bounded by computation node 1's finishing time, which is significantly higher in the suboptimal case, which uses Flow 0. Therefore, our optimal solution minimizes the finishing time for both computation nodes and chooses the flows that utilize the shared links.

5 DISCUSSION

5.1 Applications

Spark, in its current state, chooses computation worker nodes to offer based on data locality, where nodes are chosen based on their location from the data. Options for data locality include the same node, same rack in a data center, or none of these. However, within each of these categories, there is no ordering in terms of closer and further away nodes, and workers are chosen randomly. In our example topology, Spark would randomly choose computation nodes based on data location, because all computation nodes fall within the same category of not data local or rack local. This would lead to suboptimal results. With our network-aware scheduling, Spark would able to decrease the amount of time for data transfer.

As results show, SparkL is ideal for use in a weakly-connected cluster, where data transfer rates play a significant role in the total amount of time Spark jobs take to run.

5.2 Optimization Engine

As illustrated by our evaluation of the CPLEX optimization server, the time required to solve the linear programming problem increase significantly for networks with a large number of possible flows. In our simple topology, solving for the optimal set of flows is relatively fast, but in a more complicated network, this time required to solve for the optimal set of flows may exceed the benefit provided from choosing these more optimal flows if the optimization and job processing steps are synchronous.

5.2.1 Pipelining. For an initial improvement, the optimization and job processing can be pipelined so that the job processing would only need to wait on the optimization step for the initial run. Subsequent job processing can be done using the earlier optimization results. If the network bandwidth capacities are not changing rapidly, this will not show dramatic performance reduction since the optimal sets of flows would have remained the same between runs of the optimization step anyway.

In order to allow multiple sets of jobs to be run, we can also have the optimization set calculate the optimal set of flows for any number of jobs. In our implementation, we do not allow the optimal flow set to contain any duplicate uses of any computation node or storage node. Thus, in any particular step, the maximum number of jobs we can process is minimum of the number of computation nodes and the number of storage nodes. The input to our CPLEX model specifies a particular number of jobs, so in order to use the output of a single call to our CPLEX server for any number of subsequent job sets, we can run the optimization engine for all job counts from 0 to the maximum number of jobs in a single job set. With this output, we can select the optimal set of flows for any number of submitted jobs while the optimization engine computes a new sets of optimal flows in the background. SparkL can decide how often to fetch the new sets of optimal flows from the optimization engine, and does not necessarily need to wait for the optimization to complete before running the next set of jobs.

For a large number of maximum concurrent jobs, we can also calculate the optimal sets of flows for a few job counts. For example, if the maximum number of jobs in a single job set is 100, we can run the optimization for 5, 20, 50, 80, and 100 jobs. If a submitted job set contains 75 jobs, we can use the optimal flow set computed for 80 jobs and randomly choose only 75 of these flows. As a result the increase in network size does not necessarily have to correlate with a linear increase in the number of optimization runs.

5.2.2 Adaptive. Finally, as an additional refinement for large networks where running the optimization for many possible job counts and a large number of possible flows, we can implement a rudimentary learning algorithm to compensate for network bandwidth changes or cluster computation capacity changes that occur between runs of the optimization engine. After the completion of every job set, we can detect which flow resulted in the slowest job execution. If this job execution is slower by a significant margin, which can be determined based on a custom tuned threshold, we can âĂĲpunishâĂİ this flow by avoiding selecting it unless absolutely necessary. In other words, if we have optimization results for 80 jobs, and the current job set only contains 75 jobs, we can avoid selecting a flow that has been previously detected to be slow. Of course, we can still use this flow if necessary since the job set could contain 80 jobs. Also, after the next set of optimization results are received, we can clear all punished flows, since the new results will hopefully have captured this change in network or cluster state. This approach allows us to learn about slow flows incrementally over various job sets, which captures bandwidth or computation capacity changes that may occur between runs of the optimization engine.

6 FUTURE WORK

6.1 Modify Spark DAGScheduler

By modifying the Spark DAGScheduler and essentially moving our Scheduler Server into the DAGScheduler, we can link more smoothly with the existing Spark implementation. This would allow applications that currently use Spark to simply use our new compiled version of Spark, and use their existing, robust cluster management systems. At each stage of the Spark job, the DAGScheduler would call out to our CPLEX Server, receive network topology and bandwidth constraints, run an optimization, and then decide which flows to use to schedule the tasks in this stage. This would require work in decoupling dependencies that Spark adheres to, including data file locations.

6.2 Anycast

SparkL currently optimizes the selection of sets of flows, but we could also describe our problem in terms of selecting optimal storage nodes. We could design our scheduler to implement a protocol similar to the anycast addressing design. Routers in an anycast setting will receive requests for an anycast IP address and select that actual path to the destination based on certain optimization parameters.

Using a similar generic address, we could have the computation nodes all make the same request for a file to this address, but route to different storage nodes based on optimal flow selection that we've implemented. In this case, we can still select for optimal flows, but the computation nodes then do not need to have direct knowledge of the storage nodes that are available.

6.3 CPU Constraints

Our current system is network-constraint aware, however our CPLEX optimization can also be extended to take into account CPU constraints. For example, if the CPU for one of our computation nodes ends up being dramatically cut, due to another process running or a malfunction with the node, we can choose to not use that node for computation and instead use another one, in a situation where we have more nodes than jobs. To do this, we would pass in the CPU information per computation node.

Alternatively, if the CPU and network in tandem are so poor for one computation node, we could wait for another node to finish and schedule another task on it for the same stage, instead of using the poor computation node.

6.4 File Size Awareness

When multiple jobs are sent at the same time, we could probe for data file sizes before retrieving data, to optimize for using flows with large bandwidths for jobs with larger data file requirements. When the Scheduler Server receives a set of jobs, it can probe the storage node for the size of the file. Using this information as part of the optimization calculations, it can then decide which flows should serve which job and utilize which data file. For very large file size disparities between jobs, this additional bandwidth should be helpful. In some cases, this additional probing step may only add additional time for the overall job to run.

7 CONCLUSION

Our proof-of-concept system, SparkL, is shown to choose optimal flows for distributed data analytics. Based on our evaluation, we observed end-to-end performance improvements resulting from our system's optimal network bandwidth allocation. SparkL uses a global view of computation, storage, and network resources to make globally optimal decisions. When running large-scale data analytics frameworks, such as Spark, in weakly-connected networks, our network-aware system may outperform systems that do not take into account limitations resulting from network topologies. Integrating with existing systems and refining the optimization engine can allow this work to be extended to real-world, large-scale distributed applications.

8 ACKNOWLEDGEMENTS

We would like to thank Professor Richard Yang for being our senior project advisor, and for his guidance and support throughout the semester. We would also like to especially thank Jensen Zhang and Qiao Xiang for helping us immensely in troubleshooting, setting up our network, and refining our design.

9 SUPPLEMENTARY INFORMATION

Code for this project can be found at https://github.com/CS490-Knush

- `demo-utils` contains setup code for Containernet, and each server lives in is within its own repository.
- Our server APIs can be found at: Scheduler Server: https://app.swaggerhub.com/apis/cpsc490/scheduler_server/1.0.0
- CPLEX Server: https://app.swaggerhub.com/apis/cpsc490/cplex_server/1.0.0

10 REFERENCES

(1) Xiang, Q. et al. 2017. Unicorn: Unified resource orchestration for multi-domain, geo-distributed data analytics. 2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/ SCALCOM/UIC/ATC/CBDCom/IOP/SCI) (Aug. 2017).

(2) M. Peuster, H. Karl, and S. v. Rossem: MeDICINE: Rapid Prototyping of Production-Ready Network Services in Multi-PoP Environments. IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), Palo Alto, CA, USA, pp. 148-153. doi: 10.1109/NFV-SDN.2016.7919490. (2016)

(3) Shvachko, K. et al. 2010. The Hadoop Distributed File System. 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (May 2010).

(4) Zaharia, M. et al. 2010. Spark: cluster computing with working sets, Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, Boston, MA, p.10-10 (June 22-25 2010)

(5) Apache Hadoop https://hadoop.apache.org.

(6) Apache Spark https://spark.apache.org/.

(7) Agarwal, S. et al. 2018. Sincronia. Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication - SIGCOMM ’18 (2018).

(8) McKeown, N. et al. 2008. OpenFlow. ACM SIGCOMM Computer Communication Review. 38, 2 (Mar. 2008), 69. DOI: https://doi.org/10.1145/1355734.1355746.

(9) Basturk, E. et al. 1997. Using Network Layer Anycast for Load Distribution in the Internet. Tech. Rep., IBM T.J. Watson Research Center.

(10) Ousterhout, K. et al. 2013. Sparrow. Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP âĂŹ13 (2013).

(11) Senthilkumar, M. and Ilango, P. 2016. A Survey on Job Scheduling in Big Data. Cybernetics and Information Technologies. 16, 3 (Sep. 2016), 35âĂŞ51. DOI:https://doi.org/10.1515/cait-2016-0033.

(12) Chowdhury, M. et al. 2011. Managing data transfers in computer clusters with orchestra. ACM SIGCOMM Computer Communication Review. 41, 4 (Oct. 2011), 98. DOI: https://doi.org/10.1145/2043164.2018448.