Reconfigurable and Adaptive Spark Applications

Bilal Abi Farraj, Wael Al Rahal Al Orabi, Chadi Helwe, Mohamad Jaber,
Mohamad Omar Kayali and Mohamed Nassar
American University of Beirut, Computer Science Department, Beirut, Lebanon

Keywords: Big Data, Spark, Hadoop, Quality of Service, Cloud.

Abstract: The contribution of this paper is two-fold. First, we propose a Domain Specific Language (DSL) to easily reconfigure and compose Spark applications. For each Spark application we define its input and output interfaces. Then, given a set of connections that map outputs of some Spark applications to free inputs of other Spark applications, we automatically embed Spark applications with the required synchronization and communication to properly run them according to the user-defined mapping. Second, we present an adaptive quality management/selection method for Spark applications. The method takes as input a pipeline of parameterized Spark applications, where the execution time of each Spark application is an unknown increasing function of quality level parameters. The method builds a controller that automatically computes adequate quality for each Spark application to meet a user-defined deadline. Consequently, users can submit a pipeline of Spark applications and a deadline, our method automatically runs all the Spark applications with the maximum quality while respecting the deadline specified by the user. We present experimental results showing the effectiveness of our method.

1 INTRODUCTION

Since 2003, big data has become the new trend after Google started working with its project Google File System (GFS) (Ghemawat et al., 2003), and now big data has become a necessity of most new technologies. Processing big data requires implicit distribution of computation and new programming paradigm. To this end, MapReduce programming model was proposed in (Dean and Ghemawat, 2004; Dean and Ghemawat, 2010) to efficiently process data on a cluster. Some of the drawbacks of MapReduce are: (1) lack of expressiveness: does not support iterative jobs and only support two functions `map` and `reduce`; (2) not interactive: only used for batch applications; (3) efficiency: in order to implement iterative applications, intermediate results must be written and replicated on the disk (i.e., through Hadoop Distributd File System - HDFS). For instance, machine learning algorithms cannot efficiently benefit from MapReduce since they require iterative jobs to be executed. For this, Spark was proposed in (Zaharia et al., 2010) to allow interactive and iterative jobs. Spark uses resilient distributed datasets (RDD) which are a read-only collection of objects stored in the RAM. RDDs are big parallel collections that are distributed across the cluster. RDDs are created through parallel transformations (e.g., map, group by, filter, join, create from HDFS block). Moreover, RDDs can be cached by allowing to keep data sets of interest in Memory across operations and thus contribute to a substantial speedup. Additionally, after creating RDDs, it is possible to do analytics on them by running actions on them such as count, reduce, collect and save. Note that all operations/transformations are lazy until you run an action. Then, the Spark execution engine pipelines operations and finds an execution plan. In this paper, we consider Spark applications.

The contribution of this paper is two-fold.

- **Reconfigurable Component-based Spark.** We define a high-level specification language to develop Spark applications independently and automatically compose them. For each application, we export its input and output interfaces, then we define a configuration file to compose those interfaces. Then, applications are automatically augmented with the proper code to wait data/notification from other applications or notify/send data to other applications. This allows to simplify the development of complex system consisting of several dependent Spark applications. Moreover, for the same set of Spark applications, several systems may be built depending on the input configuration file. For instance, in the case of bioinfor-

Jaber, M., Nassar, M., Orabi, W., Farraj, B., Kayali, M. and Helwe, C.
Reconfigurable and Adaptive Spark Applications.
DOI: 10.5220/0006289901120119
In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*, pages 84-91
ISBN: 978-989-758-243-1

matics applications several ordering of pipelines on other applications may be defined (e.g., indexing, alignment, etc.).

- **Adaptive Execution of Spark Applications.** Several companies provide services to deploy and run Spark applications on the cloud (e.g., Amazon Web Services, Microsoft Azure). The pricing generally depends on the power of the allocated nodes and the execution time needed to run the applications. However, execution times may considerably vary over time as they depend on the application. Furthermore, non predictability of the underlying platform and operating systems are additional factors of uncertainty. For this, we propose a method to run a sequence of parameterized (Quality of Service) Spark applications within a specified time. Parameterized applications are those that can be augmented with an extra parameter denoting the quality level. For example, machine learning and graph analytics are good examples of parameterized applications, since their quality depend on the number of rounds (the more rounds the better quality). Consequently, using our method, cloud services can provide users with the ability to specify a deadline/price and a sequence of Spark applications to be executed using the best quality levels possible. A controller iteratively selects the best quality for each Spark application depending on the remaining deadline and time already used.

The remaining of this paper is structured as follows. Section 2 defines a language to easily compose Spark applications. Section 3 defines a method to adapt the quality levels to execute Spark applications while respecting a given deadline. Sections 4 discusses related work. Finally, Sections 5 and 6 draw some conclusions and perspectives.

2 RECONFIGURATION OF SPARK APPLICATIONS

Given a set of dependent Spark applications, we define a method to automatically compose them. Each spark application takes a set of inputs (e.g., files) and produces a set of outputs. An input can be either free or direct. A free input should be mapped to an output of a different spark application. A direct input is mapped to a direct path.

Formally, a spark application is defined as follows:

Definition 1 (Spark Application)**.** *A spark application SA is defined as a set of tuple* $(ins, outs)$*, where:*

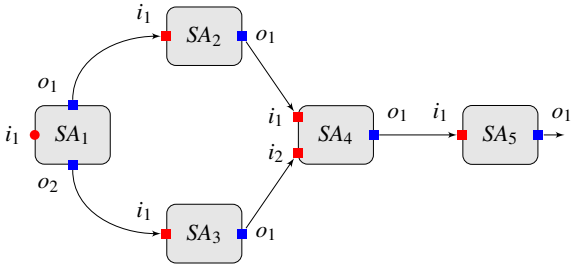

Figure 1: Representation of dependencies between subjobs.

- $ins = freeIns \cup directIns$ *is the set of inputs;*
- $outs$ *is the set of outputs.*

Given a user-specified configuration, spark applications are composed by mapping free inputs of spark applications to outputs of other applications. Formally a configuration is defined as follows:

Definition 2 (Configuration)**.** *Given a set of spark application* $\{SA_i\}_{i \in I}$*, a configuration C is a function defined by* $C : FreeInput \rightarrow Output$*, where:*

- $FreeInput = \bigcup_{i \in I} SA_i.freeIns;$
- $Output = \bigcup_{i \in I} SA_i.outs.$

Example 1. *Figure 1 shows an example of composing spark applications. The system consists of five Spark applications* $SA_1, \ldots, SA_5$*.* SA_1 *has one direct input* i_1 *and two outputs* o_1 *and* o_2*.* SA_2 *has one free input* i_1 *and one output* o_1*. First output* o_1 *of* SA_1 *is mapped to the free input* i_1 *of* SA_2*.*

Given a set of spark applications $\{SA_i\}_{i \in I}$, a configuration C must be valid. A configuration is valid iff:

- Free inputs are mapped to outputs of different applications. That is, if $C(SA_i.fi_1) = SA_j.o_1$, then $i \neq j$, where fi_1 is a free input in SA_i and o_1 is an output in SA_j.
- The graph directed G obtained from $\{SA_i\}_{i \in I}$ and C does not contain cycles, where the set of vertices of G represents Spark applications and the set of edges represents the mapping between them. Formally, $G = (\{SA_i\}_{i \in I}, E)$, where $E = \{(SA_i, SA_j) \mid \exists freeIn \in SA_i.freeIns \wedge out \in SA_j.outs : C(freeIn) = out\}$.

Example 2. *The system defined in Figure 1 is valid since (1) all free inputs are mapped to outputs of different applications; and (2) the graph obtained by connecting outputs to free inputs is acyclic.*

2.1 Semantics and Code Generation

Given a set of Spark applications and a configuration C, we first check the validity of configuration C, then

we automatically synthesize the glue of each Spark application that corresponds to configuration C. A glue surrounds each Spark application by the corresponding synchronization and communication primitives. The code generation is depicted in Listing 1. It mainly consists of the following five steps:

Listing 1: Automatic Glue Generation of a Spark Application.

```
Step 1: Fill Inputs
for(input ∈ SAi.ins) {
   if( isFreeInput(input) ) {
      source = input.from;
      index = input.index;
      inputs.add(source.getInput(index));
   } else {
      inputs.add(input.value);
   }
}

Step 2: Wait Signals
for(input ∈ SAi.freeIns) {
   waitSignal(input.from);
}

Step 3: Fill Outputs
for(output ∈ SAi.outs) {
   outputs.add(output);
}

Step 4: Run Spark Application
run("path", inputs, outputs);

Step 5: Send Signals
for(output ∈ SAi.outs) {
   free-inputs = C^-1(output)
   for(free-input ∈ free-inputs) {
      sendSignal(free-input.id);
   }
}
```

1. Fill input: inputs are mapped to their corresponding paths. In case of a free input, an output of different Spark application is filled. In case of direct input a direct path is filled.
2. Wait signals: Spark applications with free inputs $freeIns$ wait for signals from other Spark applications with outputs mapped to $freeIns$. This phase may require a setup phase to set the connections between processes (e.g., sockets, shared semaphores, signals).
3. Fill outputs: outputs are mapped to their corresponding paths.
4. Run: Run Spark application.
5. Send signals: Upon completion, Spark applications, with outputs mapped to free inputs $freeIns$, send signals to the Spark applications corresponding to $freeIns$.

2.2 DSL Implementation

We define a Domain Specific Language (DSL) using JSON representation that defines Spark applications and a configuration to connect them. Each Spark application is identified by an identifier *id*, a path where the Spark program exists *path*, number of inputs *ni*, and number of outputs *no*. Then, the configuration maps inputs to direct locations or to output of other Spark applications (in case of free inputs). Listing 2 depicts the general structure to specify a set of Spark applications and a configuration. It mainly consists of two parts:

- The first part defines the set of Spark applications with their corresponding interfaces (i.e., an identifier, location, number of inputs and number of outputs).
- The second part defines the configuration which connects Spark applications (i.e., connect output to free inputs).

Listing 2: General shape of a configuration file.

```
{"spark-applications":[
   {"id":"id", "path":"path","ni":"n","no":"n"},
   {"id":"id", "path":"path","ni":"n","no":"n"},
   ...
   {"id":"id", "path":"path","ni":"n","no":"n"},
]}

{"configuration":[
   { "id":"id",
     "i":["i1", "i2", ...],
     "o":["o1", "o2", ...],
   },
   { "id":"id",
     "i":["i1", "i2", ...],
     "o":["o1", "o2", ...],
   },
   ...
   { "id":"id",
     "i":["i1", "i2", ...],
     "o":["o1", "o2", ...],
   }
]}
```

Listing 3 shows an example of system that corresponds to Figure 1. It consists of two parts: (1) spark applications, and (2) configuration. Spark application SA_1 has one input, which is a direct input, and two outputs. Spark application SA_2 has one input and one output. The first input is mapped to the output of Spark application SA_1 (i.e., `#SA1.o1`).

Listing 3: Configuration File Corresponding to Figure 1.

```
{"spark-applications":[
   {"id":"SA1", "path":"path1","ni":"1","no":"2"},
   {"id":"SA2", "path":"path2","ni":"1","no":"1"},
   {"id":"SA3", "path":"path3","ni":"1","no":"1"},
   {"id":"SA4", "path":"path4","ni":"2","no":"1"},
   {"id":"SA5", "path":"path5","ni":"1","no":"1"},
]}

{"configuration":[
   { "id":"SA1",
     "i":["pathin1"],
     "o":["patho1", "patho2"],
   },
   { "id":"SA2",
     "i":["#SA1.o1"],
     "o":["patho1"],
   },
   { "id":"SA3",
     "i":["#SA1.o2"],
     "o":["patho1"],
   },
   { "id":"SA4",
     "i":["#SA2.o1", "#SA3.o1"],
     "o":["patho1"],
   },
   { "id":"SA5",
     "i":["#SA4.o1"],
     "o":["patho1"],
   }
]}
```

3 REAL-TIME ADAPTIVE QoS OF SPARK APPLICATION

It is difficult to predict the execution time of a set of Spark applications and can vary between two different runs. Moreover, deploying a set of Spark applications on the cloud can be very expensive and can vary on-demand, i.e., depending on the needed time. In this section, we propose a novel technique that allows users to submit a set of Spark applications with a deadline to execute all of them. For this, we consider quality-based Spark applications, i.e., each Spark application can be executed using different quality levels. We build a controller that adaptively selects the maximum quality levels for each Spark application while respecting the given deadline. This approach would be a great fit for Quality of Service applications. For the sake of simplicity, we consider a subset of CBS where Spark application defines a pipeline (i.e., the output of Spark application i is connected to the input of Spark application $i+1$).

Quality of Service (QoS) applications such as machine learning and graph analytics, iteratively process the data. That is multiple rounds of computation are performed on the same data. Therefore, the number of rounds can be easily tuned to achieve a better quality of the end result. In general, the quality is proportional to the number of rounds, and the execution times of applications are unknown increasing functions with respect to quality level parameters (i.e., increase the quality implies increase of execution time). In case of big data, these applications can be easily and efficiently implemented using Spark which provides: (1) scalability; (2) implicit distribution; (3) fault tolerance.

We introduces the adaptive method for QoS management of sequence of Spark applications. Given a sequence of parameterized spark applications and a deadline to be met, the method allows adapting the behavior of each application by adequately setting its quality level parameter while respecting the given deadline. The objective is to not miss the deadline and to select the maximum quality levels.

We consider Spark applications for which it is possible, by using timing analysis and profiling techniques, to compute estimate of worst-case execution times and average execution times for different quality levels. Worst case and average execution times will be used to estimate the execution times of Spark applications, and hence select the best quality while not exceeding the deadline.

We implement a controller that iteratively takes as input the current Spark application SA_i to be executed and the actual time[1] of the previously executed Spark applications, and it selects the quality level q to execute SA_i so that we would be able to execute the remaining applications while not exceeding the deadline. Then, the new deadline will be computed with respect to the actual execution time of $SA_i(q)$. The controller must maximize the overall quality levels. For this, it may follow several strategies. For instance, it may select the maximum quality level of the current application, however, this will end up by giving priority for first applications (i.e., selecting maximum quality) and low priority last applications (i.e., selecting minimum quality). Another option is to make the selected quality levels smooth (i.e., low deviation of quality levels with respect to the average quality). Consequently, we implement three different policies that are supported by the controller: (1) safe; (2) simple; (3) mixed.

[1]Note that the actual execution times will be only known after executing the application and will be used to adapt the remaining deadline.

3.1 Controller - Proposed Policies

We consider parameterized Spark applications, i.e., their execution times depend on the input quality. Moreover, non-predictability of the underlying cluster is another factor of uncertainty. For this, we consider that the execution times are unknown but bounded. A parameterized application can be defined as follows.

Definition 3 (Parameterized Spark System)**.** *A parameterized Spark system PSS is a tuple* $PSS = (SA, av, wc, D)$*, where*

- SA *is a sequence of Spark applications* $SA = [SA_1, \ldots, SA_n]$.
- $t^{av} : SA \times Q \to \mathbb{R}^+$*, is a function that returns the average execution time,* $t^{av}(SA_i, q)$ *for a given Spark application* SA_i *and a quality level* q. $Q = [q_{min}; q_{max}]$ *is a finite interval of integers denoting quality levels. We assume that, for all* $SA_i \in SA$, $q \mapsto t^{av}(SA_i, q)$ *is a non-decreasing function.*
- $t^{wc} : SA \times Q \to \mathbb{R}^+$*, is a function that returns the worst-case execution time,* $t^{wc}(SA_i, q)$ *for a given Spark application* SA_i *and a quality level* q*. We assume that, for all* $SA_i \in SA$, $q \mapsto t^{wc}(SA_i, q)$ *is a non-decreasing function.*
- $D \in \mathbb{R}^+$ *is the global deadline to be met.*

Let t_{i-1} be the actual execution time to execute $SA_1, \ldots, SA_{i-1}$. That is, the time to execute the remaining Spark applications (i.e., $SA_i, \ldots, SA_n$) is $D - t_{i-1}$. The controller must select the maximum quality levels for the remaining Spark applications while respecting the new deadline. We define t_e^x to be an estimation of the execution time of a sequence of Spark applications for a given quality, $t_e^x([SA_i, \ldots, SA_n], q)$. Since our goal is not to exceed the deadline, our estimation should be an over-approximation (i.e., based on worst-case execution times) of the actual execution time. We distinguish several policies to estimate execution times of the remaining applications.

Safe Policy (t_e^{sf})***:*** this defines an obvious policy to ensure safety (deadline is met) but gives more priority/time (i.e., higher quality) to first applications. Given a sequence of Spark applications and a quality level q, t_e^{sf} is defined as follows: $t_e^{sf}([SA_i, \ldots, SA_n], q) = t^{wc}(SA_i, q) + \sum_{k=i+1}^{n} t^{wc}(SA_k, q_{min})$.

Simple Policy (t_e^{sp})**:** this defines another policy to ensure safety and improves smoothness (distributed quality over all the remaining applications) by combining worst-case and average case execution times. t_e^{sp} is defined as follows: $t_e^{sp}([SA_i, \ldots, SA_n], q) = \texttt{Max}\left\{t_e^{sf}([SA_i, \ldots, SA_n], q), t_e^{av}([SA_i, \ldots, SA_n], q)\right\}$, where $t_e^{av}([SA_i, \ldots, SA_n], q) = \sum_{k=i}^{k=n} t^{av}(SA_i, q)$.

Mixed Policy (t_e^{mx})***:*** the mixed policy defines a generalization of the simple policy to improve smoothness by taking into account all possible sequences. t_e^{mx} is defined as follows: $t_e^{mx}([SA_i, \ldots, SA_n], q) = \texttt{Max}_{k=i-1}^{k=n} \{t_e^{av}([SA_i, \ldots, SA_k], q) + t_e^{sf}([SA_{k+1}, \ldots, SA_n], q)\}$.

Theorem 1. *Given a pipeline of Spark applications* $[SA_1, \ldots, SA_n]$ *and a deadline* D*. The controller will not exceed* D *for all the defined policies (i.e., Safe, Simple and Mixed) provided that the worst execution time is always greater than the actual time.*

Proof. For a given quality q, the mixed policy has a higher over-estimation of the execution time than the simple policy. Moreover, the latter has a higher over-estimation of the execution time than the safe policy. Formally, the above policies satisfy the following property: $t_e^{mx}([SA_i, \ldots, SA_n], q) \geq t_e^{sp}([SA_i, \ldots, SA_n], q) \geq t_e^{sf}([SA_i, \ldots, SA_n], q)$.

Moreover, as the safe policy is based on worst-case execution times, it clearly satisfies the safety requirement (i.e., deadline not exceeded at all iterations). Consequently, all the policies satisfy the safety requirement. □

3.2 Controller - Quality Manager Algorithm

The controller as depicted in Figure 2 must iteratively select maximum quality levels (optimality) while guaranteeing safety (deadline must be met). For this, at each iteration i, it selects (depending on the policy used) the quality level of the next Spark application according to the following:

$$\texttt{Max}\{q \mid t_e^x([SA_i, \ldots, SA_n], q) + t_{i-1} \leq D\}$$

Where t_{i-1} is the actual execution time after executing $SA_1, \ldots, SA_{i-1}$, SA_i is the current application to be executed and t_e^x is an estimation of the execution time to execute the reaming applications, i.e., $SA_i, \ldots, SA_n$. This estimation can be computed using one of the three policies defined in Subsection 3.1 (i.e., average, simple, mixed).

Example 3. *Given a sequence of three Spark applications* SA_1, SA_2, SA_3 *with deadline equals to* 9*, where the average (avet), worst case (wcet) and the actual (acet) execution times are given in the following table.*

quality	*wcet*	*avet*	*acet*
1	*1*	*1*	*1*
2	*5*	*2*	*2*
3	*6*	*3*	*3*
4	*7*	*4*	*4*

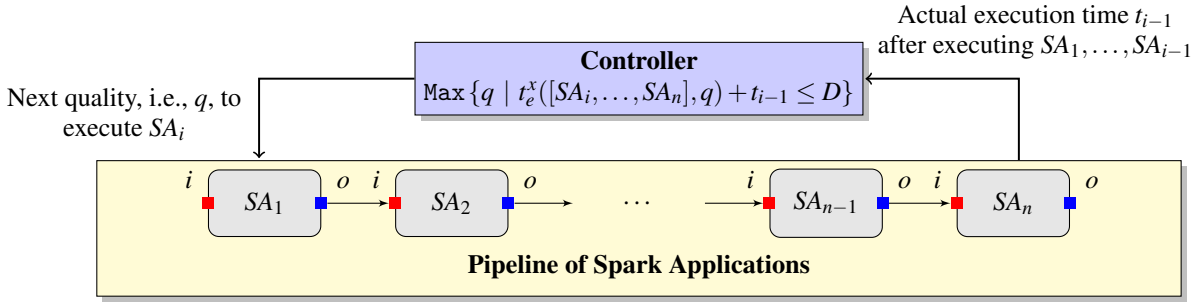

Figure 2: Adaptive Spark.

- *In case of safe policy, the quality levels* $4,1,1$ *will be selected for applications* SA_1, SA_2, SA_3*, respectively.*
- *In case of simple policy, the quality levels* $3,2,1$ *will be selected for applications* SA_1, SA_2, SA_3*, respectively.*
- *In case of mixed policy, the quality levels* $2,2,2$ *will be selected for applications* SA_1, SA_2, SA_3*, respectively.*

3.3 Experimental Results

We consider three built-in Spark applications using MLlib (Meng et al., 2015; Spark MLlib, 2016) (1) K-means clustering; (2) Logistic Regression; and (3) Support Vector Machine (SVM). We adapt these application to run in five different quality levels from 1 to 5 (highest quality). Each quality represents a specific number of iterations (higher quality more iterations). For example, quality level 4 and 5 in case of SVM correspond to 50 and 100 iterations, respectively.

We compute the average and worst-case execution times by running several benchmarks. Tables 1 and 2 depicts the average and worst-case execution times for specific number of instances and features. Note that these execution times depend on the parameters of each algorithm (e.g., number of features, size of training set). For this, in the general case, the average and worst-case execution times are functions with respect to input parameters. For the sake of simplicity, we only consider fixed value of parameters. Moreover, we consider that the deadline is greater than the summation of worst case execution times of the lowest quality.

We test our implementation by considering a sequence of seven Spark applications: SA_1 (SVM), SA_2 (Logistics Regression), SA_3 (K-means), SA_4 (Logistics Regression), SA_5 (K-means), SA_6 (SVM) and SA_7 (SVM). The deadline is 1000 seconds. Figure 3 depicts the quality levels selected by the controller using safe, simple and mixed policies, respectively. Clearly,

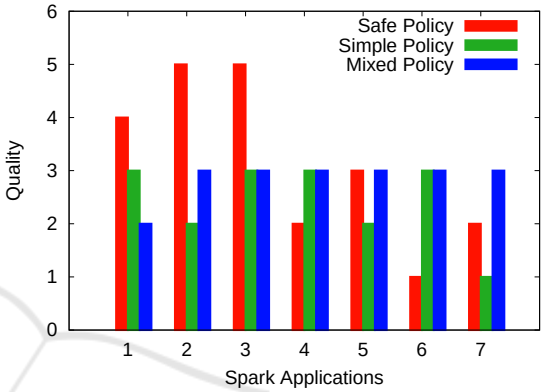

Figure 3: Selected Quality by the Controller using Safe, Simple and Mixed Policies.

the quality levels selected using the safe policy gives more priority to first applications. The simple policy provides a small smoothness of quality levels. The mixed policy provides a fair distribution (good smoothness) of quality levels between all applications.

4 RELATED WORK

4.1 Data-Parallel Pipeline

Cascading (Cascading, 2016) and FlumeJava (Chambers et al., 2010) are Java libraries that simplify the development of data-parallel pipelines. Unlike our method: (1) the proposed frameworks/libraries are not compatible with Spark applications; (2) they allow pipelines/graph modeling within an application and not between applications. Note that, our method can be combined with those frameworks to give more composition expressiveness.

4.2 Large Scale Graph Processing

Pregel (Malewicz et al., 2010) and Giraph (Giraph, 2016) are systems for large scale graph process-

Table 1: Worst case running time.

Algorithms	Q1	Q2	Q3	Q4	Q5
K-means	115s	125s	135s	140s	145s
Logistic Regression	120s	135s	140s	150s	155s
SVM	130s	170s	190s	235s	345s

Table 2: Average case running time.

Algorithms	Q1	Q2	Q3	Q4	Q5
K-means	95s	115s	125s	130s	135s
Logistic Regression	110	125s	130s	145s	150s
SVM	90s	140s	160s	205s	315s

ing. The two systems are inspired by the Bulk Synchronous Parallel BSP model (Valiant, 2011). Such systems take as input a directed graph and they define programs to express a sequence of iterations (receive messages, send messages, local computation) depending on the vertex identifier. Our approach differs from this work in the following aspects. First, they are not applicable to Spark. Second, they are only specific to graph processing. Note also that our approach can be applied on top of these systems.

4.3 High-Level Specific Big Data Library

Several high-level specific libraries were proposed to process big data. For example, MLlib (Meng et al., 2015; Spark MLlib, 2016) provides machine learning library on top of Spark. Apache Mahout (Mahout, 2016) allows to build an environment for quickly creating machine learning applications. Our high-level specific language allows to compose different applications written using different high-level libraries. Thus, developers can easily build a complex system consisting of several applications using several libraries.

4.4 Adaptive QoS

In (Wust et al., 2004; Buttazzo et al., 1998; R.I.Davis et al., 1993), they proposed solutions to achieve a soft real-time execution multimedia applications based on several techniques such as Markov decision process, reinforcement learning and Earliest Deadline First (EDF).

In (Combaz et al., 005b), they proposed a method where the quality levels can be set adequately in case of multimedia applications so that the following QoS requirements are respected: (1) Safety - No deadline is missed; (2) Optimality - Maximize the available time; and (3) Smoothness of quality levels. Our method is a variation of the method presented in (Combaz et al., 2008) where we apply it on a sequence of Spark applications. To the best of our knowledge, we have not seen major work that allows the cloud to be parameterized to dynamically allocate maximum quality levels for a sequence of submitted jobs while respecting a user-input deadline (i.e., price/cost).

4.5 Dynamic Partitioning

In (Gounaris et al., 2017), they introduce an algorithm for dynamic partitioning of RDDs with a view to minimizing resource consumption. The algorithm uses analytical models for expressing the running time as a function of the number of machines employed. Unlike our method, it allows to dynamically select quality levels for a whole Spark application. Note that the two methods may be combined to tune the granularity of dynamism.

5 CONCLUSION

We have presented a new approach for linking Spark applications to build a complex one based on a user-defined configuration file. Given a set of Spark applications, a configuration file defines a directed graph between the applications, where each edge is a dependency between two applications. Dependency denotes order of execution and data dependency (e.g., the input of an application requires the output of a different application). We have implemented a Domain Specific Language (DSL) to easily define interfaces as well as connections between them. Our framework automatically build the final system with respect to the input configuration.

Furthermore, we have defined a method that allows clusters to be augmented with controllers in order to automatically manage quality levels of a sequence of Spark applications. Thus, our method allows users to run a set of applications with the best

possible quality without violating time constraints (e.g., deadline). We discussed several policies in order to either give more priority for early applications or to obtain a fair quality between all the applications.

6 FUTURE WORK

The future work goes into several directions. First, we consider to define a unified DSL that combines several frameworks and not only Spark. Second, we will introduce interface place holders within a Spark application to allow more expressive composition. Third, we will support several clusters and hence efficiently transfer data at synchronization points become a challenging task. Another direction is to extend the controller to take as input a graph of Spark applications (not only a sequence). For this, the controller should be augmented with a scheduler to select the best next Spark application in addition to the next quality level.

REFERENCES

Buttazzo, G. C., Lipari, G., and Abeni, L. (1998). Elastic task model for adaptive rate control. *RTSS*, pages 286–295.

Cascading (2016). Cascading. http://www.cascading.org.

Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R. R., Bradshaw, R., and Weizenbaum, N. (2010). Flumejava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 363–375.

Combaz, J., Fernandez, J., Sifakis, J., and Strus, L. (2008). Symbolic quality control for multimedia applications. *Real-Time Systems*, 40(1):1–43.

Combaz, J., Fernandez, J.-C., Lepley, T., and Sifakis, J. (2005b). Qos control for optimality and safety. *Proceedings of the 5th Conference on Embedded Software*.

Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150.

Dean, J. and Ghemawat, S. (2010). Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77.

Ghemawat, S., Gobioff, H., and Leung, S. (2003). The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 29–43.

Giraph (2016). Apache giraph. http://giraph.apache.org.

Gounaris, A., Kougka, G., Tous, R., Tripiana, C., and Torres, J. (2017). Dynamic configuration of partitioning in spark applications. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1.

Mahout (2016). Apache mahout. http://mahout.apache.org.

Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146.

Meng, X., Bradley, J. K., Yavuz, B., Sparks, E. R., Venkataraman, S., Liu, D., Freeman, J., Tsai, D. B., Amde, M., Owen, S., Xin, D., Xin, R., Franklin, M. J., Zadeh, R., Zaharia, M., and Talwalkar, A. (2015). Mllib: Machine learning in apache spark. *CoRR*, abs/1505.06807.

R.I.Davis, K.W.Tindell, and A.Burns (1993). Scheduling slack time in fixed priority pre-emptive systems. *Proceeding of the IEEE Real-Time Systems Symposium*, pages 222–231.

Spark MLlib (2016). Spark mllib. http://spark.apache.org/mllib/.

Valiant, L. G. (2011). A bridging model for multi-core computing. *J. Comput. Syst. Sci.*, 77(1):154–166.

Wust, C. C., Steffens, L., Verhaegh, W. F., Bril, R. J., and Hentschel, C. (2004). Qos control strategies for high-quality video processing. *Euromicro Conference on Real-Time Systems*, pages 3–12.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*.