Future Generation Computer Systems ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Fine and coarse grained composition and adaptation of spark applications

Zeinab Shmeis, Mohamad Jaber [*]

American University of Beirut, Computer Science Department, Lebanon

ARTICLE INFO

Article history:
Received 15 September 2017
Received in revised form 26 March 2018
Accepted 17 April 2018
Available online xxxx

Keywords:
Spark
Component-based design
Quality of service
Code generation

ABSTRACT

Spark is a framework used to analyze big data applications. In this paper, we introduce a framework to build complex Spark applications by composing simpler ones. We use two levels of granularity for composition. The fine (resp. coarse) granularity focuses on composing sub-Spark (resp. Spark) applications to build a more complex one. Composition takes as input a configuration file that defines the connection between sub-spark and Spark applications. Moreover, in case of composing sub-Spark applications, we introduce different scenarios to automatically persist and un-persist most used data to achieve a better performance. We also present a method to parameterize a system consisting of several Spark applications with respect to their quality of executions. Then, we introduce several strategies to dynamically select the maximum quality levels to execute the given Spark applications, while meeting a user-defined deadline. We present experimental results showing the effectiveness of our method with respect to composition, performance and quality of service of Spark applications.

1. Introduction

Since 2003, Big Data has become the new trend after Google started working with its project Google File System (GFS) [1], and this huge interest in Big Data has advanced the development of systems for large scale analytics. The MapReduce programming model proposed in [2,3] was the first to efficiently process data on a cluster. However, MapReduce has several limitations: (1) it introduces latency between each map and reduce step because of shuffling its data to the network and writing them to the disk; (2) it is not interactive as it can only process data in batches; (3) it is unable to support iterative applications, which are required by most algorithms in data science and machine learning toolboxes; (4) it is not user friendly as it can only support two functions **map** and **reduce**. For this, Spark was proposed in [4] to allow interactive and iterative jobs and after that evolved to be one of the most active open-source Apache projects with a huge developer and user community. Spark abstracts data collections as Resilient Distributed Datasets (RDDs), which are partitioned across several nodes so that they can be operated on in parallel. The Spark programming model is easy to use and have extensible APIs in different languages such as Scala, Java, Python, and R. It is based on higher-order functions that execute user-defined functions. These higher order functions are of two types: Transformations and Actions. Transformations, such as **map** and **filter**, apply a function on each RDD element and result in a new RDD. Actions trigger the execution of such functions and produce meaningful results which is returned to the coordinator program, called driver. For each action in a Spark application, a job is performed, which includes several RDD transformations.

Spark's scheduler creates a physical execution plan for the job based on the RDD lineage (aka RDD dependency graph) which is a directed acyclic graph (DAG) of transformations. The physical plan is divided into stages. A stage is a sequence of transformations that can be pipelined (executed, without data movement, in parallel over all data partitions). Pipelining transformations are deliberately grouped together in a single stage to speed-up performance. The sequence of computations defined by a stage instantiated over a single data partition is called a task. A task is the actual unit of execution of the physical plan.

The contribution of this paper is two-fold.

- **Component-based Spark.** Spark applications are getting larger and more complex, calling for a shift from Spark programming to Spark system composing. Therefore, 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. Our language handles two cases of composing Spark applications. The first case, coarse grained composition, each Spark application is considered as one component that can run as a whole and writes its output

[*] Corresponding author.
E-mail addresses: zhs07@mail.aub.edu (Z. Shmeis), mj54@aub.edu.lb (M. Jaber).

https://doi.org/10.1016/j.future.2018.04.048
0167-739X/© 2018 Elsevier B.V. All rights reserved.

to distributed file system that can be later on read by another Spark application as input. In this case, our method automatically augments each application with the proper code to send (e.g., when a Spark application terminates) and receive from/to other applications. Whereas the other case, fine grained composition, we consider sub-Spark applications, which can be composed to generate a more complex one. Each sub-Spark application consists of a sequence of computation blocks separated with place holders denoting input and output RDDs or location in a distributed file system. As such, a mapping can be defined between the place holders of sub-Spark applications. Then, our method automatically merges sub-Spark applications by renaming variables and ordering blocks of executions, and generates a monolithic Spark application. Moreover, for the fine grained composition, we define a method to persist outputs that are used by several other sub-Spark application in order to improve the overall execution time of the new resulting program. These two cases of composition allows to simplify the development of complex system consisting of several dependent Spark applications. Furthermore, for the same set of (sub-)Spark applications, several systems may be built by simply changing the input configuration file.

- **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.

This paper is an extended version of our earlier work in [5]. More specifically, this paper provides the following additional contributions: (1) integrating the fine grained composition of sub-Spark applications and distinguishing it from the coarse grained composition defined in the previous work, (2) defining several strategies to automatically (un)persist data for the fine-grained composition, (3) extending the experimental result work by adding new case studies, in addition to (4) extending the related work.

The remaining of this paper is structured as follows. Section 2 defines the language for the coarse grained composition of Spark applications. Section 3 describes the fine grained composition of sub-Spark applications, and defines the strategies used for (un)persisting data. Section 4 defines a method to adapt the quality levels to execute Spark applications while respecting a given deadline. Section 5 discusses related work. Finally, Sections 6 and 7 draw some conclusions and perspectives.

2. Coarse grain composition of Spark applications

Given a set of dependent Spark applications, we define a coarse grain composition, which automatically generates the wait/receive and notify/send setup between them. Each spark application takes a set of inputs (e.g., location to files) and produces a set of outputs (files). We distinguish between free and direct input. A free input is an input mapped to an output of a different spark application, while a direct input is a one mapped to a direct path. Formally, a spark application is defined as follows:

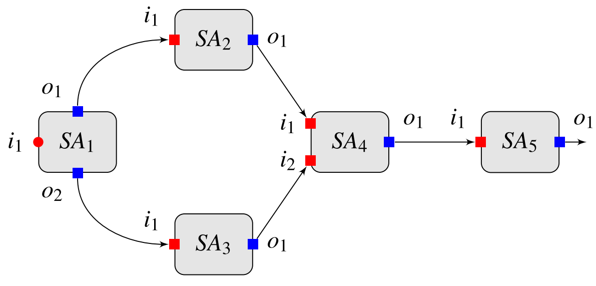

Fig. 1. Representation of dependencies between sub-jobs.

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

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

Given a user-specified configuration, spark applications are composed by mapping outputs of spark applications to free inputs 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 $C : Input \rightarrow Output$, where:

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

Example 1. Fig. 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.

Definition 3. Given a set of spark applications $\{SA_i\}_{i\in I}$, and a configuration C, we define the directed graph $G = (V, E)$, where:

- $V = \{SA_i\}_{i\in I}$, is the set of vertices representing the Spark applications in the set;
- $E = \{(SA_i, SA_j) \mid \exists\, out \in SA_i.outs \wedge freeIn \in SA_j.freeIns : C(freeIn) = out\}$, is the set of edges representing the mapping between $\{SA_i\}_{i\in I}$.

Based on the build graph from $\{SA_i\}_{i\in I}$ and C, 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 directed graph G obtained from $\{SA_i\}_{i\in I}$ and C does not contain cycles, that is no vertex is reachable from itself.

Example 2. The system defined in Fig. 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.