A SURVEY OF BIG DATA FRAMEWORKS ON A LAYERED DATAFLOW MODEL

Claudia Misale, Maurizio Drocco, Marco Aldinucci and Guy Tremblay

May 2016

Département d'informatique

Université du Québec à Montréal

Rapport de recherche Latece 2016-1

Laboratoire de recherche sur les technologies du commerce électronique

A SURVEY OF BIG DATA FRAMEWORKS ON A LAYERED DATAFLOW MODEL

Claudia Misale
Comp. Sc. Dept.
University of Torino
Torino, Italy

Maurizio Drocco
Comp. Sc. Dept.
University of Torino
Torino, Italy

Marco Aldinucci
Comp. Sc. Dept.
University of Torino
Torino, Italy

Guy Tremblay
Dép. d'informatique
UQAM
Montréal, Qc, Canada

Laboratoire de recherche sur les technologies du commerce électronique
Département d'informatique
Université du Québec à Montréal
C.P. 8888, Succ. Centre-Ville
Montréal, QC, Canada
H3C 3P8
http://www.latece.uqam.ca

May 2016

Rapport de recherche Latece 2016-1

This work has been partially supported by the EU FP7 REPARA project (no. 609666), the EU H2020 Rephrase project (no. 644235) and the 2015–2016 IBM Ph.D. Scholarship program.

Abstract

In the world of Big Data analytics, one can find a series of tools aiming at simplifying programming applications to be executed on clusters. Although each tool is claiming to provide better programming, data and execution models—for which only informal (and often confusing) semantics is generally provided—all share a common underlying model, namely, the Dataflow model. The Dataflow model we provide shows how various tools share the same expressiveness at different levels of abstraction. The contribution of this work is twofold: first, we show that the resulting model is (at least) as general as existing batch and streaming frameworks (e.g., Spark, Flink, Storm), thus making it easier to reason about high-level data-processing applications written in such frameworks. Second, we provide a layered model that can represent tools and applications following the dataflow paradigm and we show how the analyzed tools fit in each level.

1 Outline

With the increasing number of Big Data analytics tools, we assist at a continuous fight among implementors/vendors in demonstrating how their tools are better than others in terms of performances and/or expressiveness. In this hype, for a user approaching Big Data analytics (even an educated computer scientist), it might be hard to have a clear picture of the programming model underneat these tools and the expressiveness they provide to solve some user defined problem. In this scenario, we aim to understand exact features those tools provide to the user in terms of API and how they are related to parallel computing paradigms. To provide some order in the world of Big Data processing, in this paper we categorize models and tools to extract common features in their programming models.

At this aim, we identified the *Dataflow model* [15] as the common model that better describes all levels of abstraction, from the user-level API to the execution model. This model represents applications via a directed graph of actors. In its “modern” reissue (a.k.a. macro-data flow [3]), it naturally models independent (thus parallelizable) kernels starting from a graph of true data dependencies. Kernels execution is triggered by data availability.

The Dataflow model is expressive enough to describe batch, micro-batch and streaming models that are implemented in most tools for Big Data processing. Being all realized under the same common idea, we show how various Big Data analytics tools share almost the same base concepts, differing mostly in their implementation choices. We instantiate the Dataflow model into a stack of layers where each layer represents a dataflow graph/model with a different meaning, describing a program from what the programmer sees down to the underlying, lower-level, execution model layer. Furthermore, we put our attention to a problem rising from the high abstraction provided by the model that reflects into the examined tools. Especially when considering stream processing and state management, non-determinism may rise when processing one or more streams in one node of the graph, which is a well-known problem in parallel and distributed computing. Eventually, the paper focus on high-level parallelism exploitation paradigms and the correlation with Big Data tools at the level of programming and execution models.

In this paper, we examine the following tools under the dataflow perspective: Spark [18], Storm [16], Flink [1], and TensorFlow [2]. As far as we know, no previous attempt was made to compare different Big Data processing tools, at multiple levels of abstraction, under a common formalism.

The paper proceeds as follows. Sections 2 and 3 describe the Dataflow model and how it can be exploited at three different abstraction levels. Sec. 4 focuses on user-level API of the tools. The discussion about each level of our layered model is provided in Sections 5, 6 and 7. Eventually, Sections 8 and 9 discuss some limitations of the dataflow model in capturing all the tool features and frames the programming model of the tools in a historical perspective. Finally, Section 10 concludes the paper, also describing some future work.

2 Dataflow Process Networks

In this section, we review the Dataflow model of computation, as defined by Lee and Parks [15]. We underline the formal properties of the basic Dataflow model that better describes all models involved in our layered model from Fig. 1 (p. 3). We show how the model is general enough to subsume many different levels simply by changing the semantics of dataflow nodes and links.

Dataflow Process Networks are a special case of Kahn Process Networks, a model of computation that describes a program as set of concurrent processes communicating among each other via FIFO channels, where reads are blocking and writes are non-blocking. A set of *firing rules* is associated to each process, here called *actor*. In a dataflow network, processing consists of "repeated firings of actors", where an actor represents a *functional* unit of computation over *tokens*. For an actor, to be functional means that firings have no side effects (thus actors are stateless) and the output tokens are functions of the input tokens. The model can be extended to allow stateful actors.

A dataflow network can be executed in several ways. The two main classes of execution are *process-based* and *scheduling-based*—other models are flavors of these two. The process-based model is straightforward: each actor is represented by a process that communicates via FIFO channels. In the scheduling-based model, also known as dynamic scheduling, a scheduler tracks the availability of tokens in input to actors and schedules enabled actors for execution; the atomic scheduling unit is referred as a *task* and represents the computation performed by an actor over a single set of input tokens.

Actors A dataflow actor consumes input tokens when it "fires" and then produces output tokens; thus it repeatedly fires on tokens belonging to one or more streams. The function mapping input to output tokens is called the *kernel* of an actor[1]. A *firing rule* defines when an actor can fire. Each rule defines what tokens have to be available for the actor to fire. Multiple rules can be combined to program arbitrarily complex firing logics (e.g., the *If* node).

Input channels The kernel function takes as input one or more tokens from one or more input channels when a firing rule is activated. The basic model can be extended to allow for testing input channels for emptiness, in order to express arbitrary stream consuming policies (e.g., gathering from any channel), as we will remark in Sec. 8.

[1]We remark the Dataflow process network model seamlessly comprehends the Macro Dataflow parallel execution model, in which each process executes arbitrary code. Conversely, actor codes in the classical Dataflow *architecture* model are single instructions of the architecture language. From now on, we refer to both Dataflow and Macro Dataflow models with a single formalism.

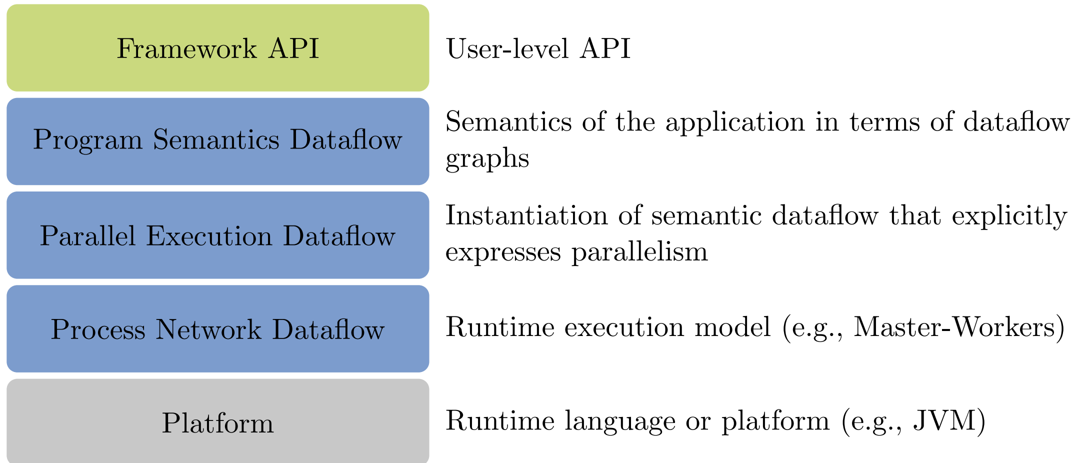


Figure 1: Layered model representing the levels of abstractions provided by the frameworks that were analyzed.

Output channels The kernel function places one or more tokens into one or more output channels when a firing rule is activated. Each output token produced by a firing can be replicated and placed onto each output channel (i.e., broadcasting) or sent to specific channel(s), in order to model arbitrarily producing policies (e.g., switch, scatter).

Stateful actors Actors with state can be considered like objects (instead of functions) with methods used to modify the object's contents. Stateful actors is an extension that allows side effects over *local* (i.e., internal to each actor) states. It is shown in [15] that stateful actors can be emulated in the functional dataflow model by adding an extra feedback channel carrying the value of the state to the next execution of the kernel function on the next element of the stream. From the architectural model perspective, this extension corresponds to a form of the so-called hybrid Dataflow-von Neumann model.

3 The Dataflow Layered Model

By analyzing some well-known tools—Spark, Storm, Flink, and TensorFlow—we were able to identify a common structure underlying all of them. We formalized an architecture that can describe all these models at different levels of abstraction, from the (top) user-level API to the (bottom-level) actual network of processes. The layered model shown in Fig. 1 presents five layers, where the three intermediate layers are Dataflow models with different semantics, as described in the paragraphs below. Underneath these three layers is the *Platform* level, that is, the runtime or programming language used to implement a given framework (e.g., Java and Scala in Spark), a level beyond the scope of our paper. On top is the *Framework API* level, that describes the user API on top of the dataflow graph and it will be detailed in Section 4. The three Dataflow models in between are as follows.

- *Program Semantics Dataflow*: We claim the API exposed by any of the considered frameworks can be translated into a dataflow graph. The top level of our layered model captures this translation: programs at this level represent the *semantics* of

data-processing applications in terms of dataflow graphs. Programs at this level do not explicitly express any form of parallelism: they only express data dependencies (i.e., edges) among the program components (i.e., actors). This aspect will be covered in Section 5.

- *Parallel Execution Dataflow*: This level, covered in Section 6, represents an effective instantiation of the semantic dataflows in terms of processing elements (i.e., actors) connected by data channels (i.e., edges). Independent units—not connected by a channel—may execute in parallel. For example, a semantic actor can be replicated to express *data parallelism*, the execution model in which a given function is applied to independent input data (i.e., tokens).
- *Process Network Dataflow*: This level, covered in Section 7, describes how the program is effectively deployed and executed onto the underlying platform. Actors are concrete computing entities (e.g., processes) and edges are communication channels. The most common approach—used by all the considered frameworks but TensorFlow—is for the actual network to be a Master-Workers task executor, where a task is generated every time an actor from the upper layer becomes enabled. In TensorFlow, processing elements are effectively mapped to threads and possibly distributed over multiple nodes of a cluster.

4 The Framework API

Data-processing applications are generally divided into *batch* vs. *stream* processing. Batch programs are designed by composing operations over one or more finite datasets, while stream programs target possibly unbounded sequences of data, called streams. Operations over streams may have to respect a total data ordering, not needed in batch computations.

We divide API expressiveness in two categories: *declarative* and *topological* data processing. As we now show, Spark and Flink belong to the first category—they provide both batch and stream processing in the form of operators over collections and streams—whereas Storm and TensorFlow belong to the second one—they provide an API explicitly based on graphs.

4.1 Declarative Data Processing

This model provides as building blocks data collections and operations on such data. The data model follows domain-specific operators, for instance, relational algebra operators that operate on data structured with the key-value model.

Batch Processing applications are implemented as methods on objects representing collections: this is an algebra on finite datasets where no data ordering is needed. APIs with such objects and transformations are exposing a functional-like style. We provide three examples of operations together with their respective semantics:

$$\begin{aligned} \texttt{groupByKey}(a) &= \{(k, \{v : (k, v) \in a\})\} && (1) \\ \texttt{join}(a, b) &= \{(k, (v_a, v_b)) : (k, v_a) \in a \wedge (k, v_b) \in b\} && (2) \\ \texttt{map}\langle f \rangle(a) &= \{f(v) : v \in a\} && (3) \end{aligned}$$

Here, the $\{\cdot\}$ syntax refers to multisets rather than sets. The `groupByKey` unary operation groups tuples sharing the same key (i.e., the first field of the tuple); thus it maps multisets of type $(K \times V)^*$ to multisets of type $(K \times V^*)^*$. The binary `join` operation merges two multisets by coupling values sharing the same key. Finally, the unary higher-order `map` operation applies the kernel function f to each element in the input multiset.

Stream processing programs are expressed in terms of an algebra on eventually unbounded data (i.e., stream as a whole) where data ordering eventually matters. Data is usually organized in tuples having a key field that determines its position in the stream. This model is fundamental to describe an order among elements in a stream by using a global timestamp as key (natural key-value model), but is used also to categorize streams into substreams (artificial key-value model). For instance, this allows relational algebra operators or data shuffling and grouping based on keys.

In a stream processing scenario, we also have to consider two important aspects: state management and windowing. Stateful computations allow maintaining some information during the whole application. State can be local to each actor, thus accessible via primitives such as set/get, or managed with functional constructs that update a local state by a kernel function. Windowing is used to organize streams into partitions obeying certain criteria (such as time intervals) and with a fixed cardinality. Windowing is a particular form of local stateful computation, where windows are created by maintaining an history over stream items.

Apache Spark API implements batch programming with a set of operators, called *transformations*, that are uniformly applied to whole datasets called *Resilient Distributed Datasets* (RDD) [18], which are immutable collections of data. For stream processing, Spark implements an extension through the Spark Streaming module, providing a high-level abstraction called *discretized stream* or DStream [19]. Such streams of data represent results in continuous sequences of RDDs of the same type, called *micro-batch*. Operations over DStreams are simply "forwarded" to each RDD in the DStream, thus the semantics of operations over streams is defined in terms of batch processing:

$$\texttt{op}(a) = [\texttt{op}(a_1), \texttt{op}(a_2), \ldots]$$

where $[\cdot]$ refers to a possibly unbounded ordered sequence, $a = [a_1, a_2, \ldots]$ is a stream (i.e., a DStream in Spark terminology), and each item a_i is a micro-batch of type RDD.

Apache Flink's main focus is on stream programming. The abstraction used is the DataStream, which is a representation of a stream as a single object. Operations are pipelined by calling operators on a DataStream objects. Apache Flink provides a data type for batch applications, called DataSet, that identifies a single collection (i.e., a stream of one element) on which to apply operators. A Flink program, either for batch or stream processing, is realized as an algebra of stateful operators over DataStream. State management will be discussed in section 4.3.

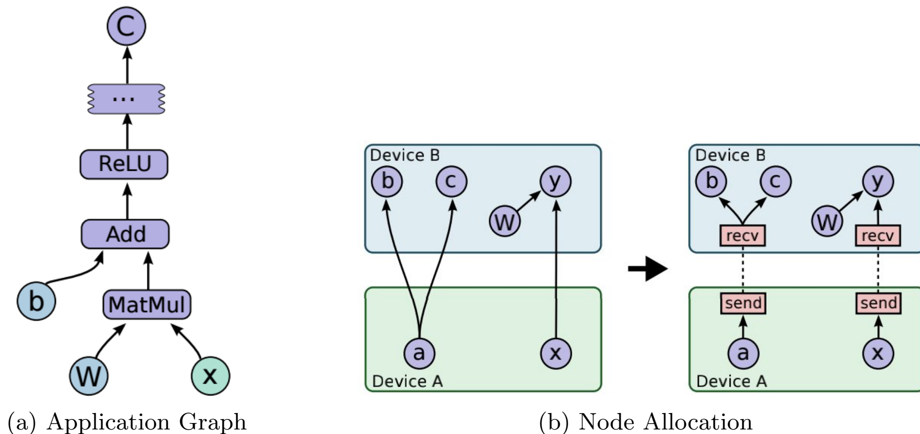
4.2 Dataflow Data Processing

Storm and TensorFlow are two tools that provide a topological API. Topological programs are expressed in terms of graphs, either explicitly (Storm) or with the description of directed

graph built by creating operators as nodes and input data as parameters of the operator (TensorFlow).

Apache Storm is a framework that only targets stream processing. Storm's programming model is based on three objects: *Spouts*, *Bolts*, and *Topologies*. A Spout is a source of a stream, that is (typically) connected to a data source or that can generate its own stream. A Bolt is a processing element, so it processes any number of input streams and produces any number of new output streams. Most of the logic of a computation goes into Bolts, such as functions, filters, streaming joins or streaming aggregations. Hence, Bolts are parametrized with per-tuple kernel code. Bolts and Spouts are stateless by default, in the sense that no operator is provided to manage state consistency. Since Bolts and Spouts are objects, stateful computations can be implemented by the user, but state management is the sole responsibility of the user, who has to ensure program correctness. A Topology is the composition of Spouts and Bolts resulting in a network. Storm uses *tuples* as its data model, that is, named lists of values of arbitrary type.

Google TensorFlow is a framework specifically designed for machine learning applications, where the data model consists of multidimensional arrays called *tensors* and a program is a directed graph representing operators processing tensors. An application graph consists in a set of *Operation* objects, which represent units of computation (or nodes), and *Tensor* objects, representing the units of data that flow between operators. We start with an example from TensorFlow whitepaper [2] in Fig. 2(a) to exemplify the programming model.



(a) Application Graph

(b) Node Allocation

Figure 2: Graph of a small TensorFlow application (a). A different example with the node distribution with Send-Receive paradigm implemented with Send/Receive nodes (b). Note: those two instances are not referring to the same example application.

Each node of the graph represents an operation, that can be also a data generation operation (nodes W, b, x). Each node has firing rules that depend on the kind of incoming tokens. For example, edges called *control dependencies* can carry synchronization tokens: the target node of such edges cannot execute its code until all appropriate synchronization signals have been received.

The TensorFlow programming model brings to the user aspects that are considered low-level, especially in frameworks for big-data applications, where the need for high-level programming is more important. In particular, the group of control flow operations contains a set of primitives to manage loops—which will be described later—and synchronization primitives. Those operations allow synchronization among nodes from the communication point of view, along with mutexes—*MutexAquire*, *MutexRelease*—used for accessing shared data (or global state). In general, a higher expressiveness increases the difficulty, for non-expert users, of writing correct programs (for instance, possible data races on global states).

4.3 State, Windowing and Iterative Computations

In this section, we examine other, transversal, aspects of the various tools, considering state management, iterative computations and windowing for stream processing.

4.3.1 State

Frameworks providing stateful processing make it possible to express modifications (i.e., side-effects) to the system state that will be visible at some future point. If the state of the system is *global*, then it can be accessed by all system components. For example, TensorFlow mutable variables are a form of global state, since they can be attached to any processing node. On the other hand, *local* states can be accessed only by a single system component.[2] For example, the `mapWithState` functional in the Spark Streaming API realizes a form of local state, in which successive executions of the functional see the modifications to the state made by previous ones. Furthermore, state can be partitioned by shaping it as a tuple space, following, for instance, the aforementioned key-value paradigm. With the exception of TensorFlow, all the considered frameworks provide local key-value states.

Constructs for working on states can be defined in the formalism provided by the framework. This is the case of TensorFlow's primitives for reading and modifying mutable variables. Alternatively, the state API can be provided as an enrichment (i.e., a library) of the language used for programming the kernel code of functionals. For example, calls to Flink's `value` and `update` functions can be inserted in kernel code for reading and writing the local state, respectively. In addition to generic load/store of states, arbitrary APIs can be designed to provide different state semantics. For example, TensorFlow provides a specific API for working on mutable queues.

From a Dataflow perspective, stateful actors represent an extension to the basic model. In particular, this extension breaks the functional nature of the basic Dataflow model, inhibiting for example to reason in pure functional terms about program semantics, as discussed in Sec. 8.

4.3.2 Windowing

A *window* is informally defined as an ordered subset of items extracted from the stream. The most common form of windowing is referred as *sliding window* and it is characterized by the size (how many elements fall within the window) and the sliding policy (how items enter and exit from the window). Spark provides the simplest abstraction for defining windows, since

[2] In principle, one could define states that are local to a subset of components (rather than a single component), but none of the considered frameworks provide this ability.

they are just micro-batches over the DStream abstraction where it is possible to define only the window length and the sliding policy. Storm and Flink also apply this kind of grouping, the former grouping Tuples and the latter grouping DataStreams. Windows in Spark are usually processed independently and out of order, thus it is not possible to store windows locally for further analysis. Storm offers a richer API by giving access to the current tuples in the window

From a Dataflow perspective, as already mentioned, windowing is not an aspect of the model itself. Windows are usually created within each actor by storing stream items into some local state [10]. Since in the analyzed frameworks, state is not used for this kind of operations, windows are created by one actor that also changes the granularity of items flowing over the network. The other actors consuming windows only have access to the current item.

4.3.3 Iterations

In Flink, iterative algorithms are defined by a *step function*, executed in each iteration, embedded into a special *iteration operator*. The step function is an arbitrary dataflow consisting of operators like `map`, `reduce`, `join`, etc. There are two types of iteration operators: `Iterate` and `Delta Iterate`. Both operators repeatedly invoke the step function on the current state until a certain termination condition is reached (maximum number of iterations or custom aggregator). Flink's iteration interface differs massively from iterative computations in Spark, since the latter does not provide any specific construct to implement iteration. The code related to transformations on data is embedded into sequential code, and the resulting dataflow is a certain number of repetitions of the transformations present into the loop.

TensorFlow allows expressing conditionals and loops as specific control flow operators.

5 Program Semantics Dataflow

This level of our layered model provides a dataflow representation of the program semantics. Such a dataflow model describes the application in terms of operators and data dependencies among them. This level does not explicitly express parallelism: instead, parallelism is *implicit* through the data dependencies among actors (i.e., among operators), so that operators which have no direct or indirect dependencies can be executed concurrently.

5.1 Semantic Dataflow Graphs

A semantic dataflow graph is a pair $G = \langle V, E \rangle$ where vertexes V represent operators and edges E represent data dependencies among operators. Firing rules can be designed to model from-any or from-all consuming nodes. In the first case, the actor is activated once it receives one input token from any input channel, thus enabling a non-deterministic behavior in its firing rule (i.e., on input choosing). Whereas in the second case the actor needs all inputs to be activated. In all the considered frameworks, output tokens are broadcast onto all channels going out of a node.

For instance, consider a map function m followed by a reduce function r on a collection of data A and its result b, represented as the functional composition $b = r(m(A)))$. This is

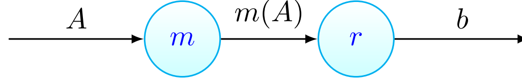
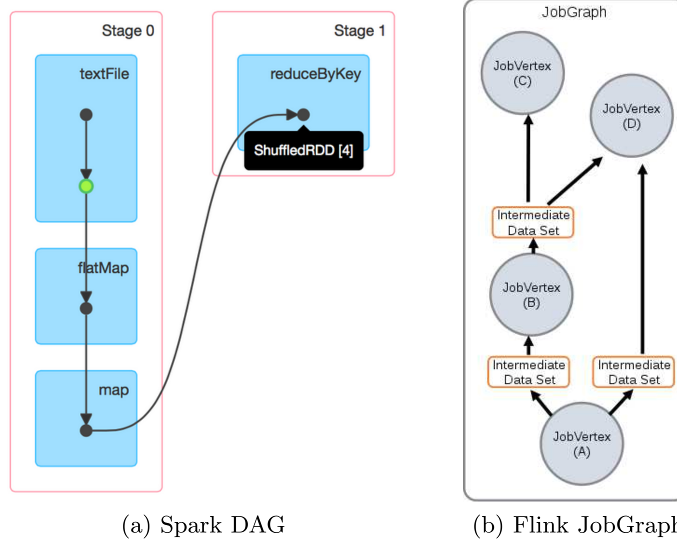


Figure 3: Functional Map and Reduce dataflow expressing data dependencies.



(a) Spark DAG (b) Flink JobGraph

Figure 4: Spark WordCount semantics dataflow identified by the application DAG (a). Apache Flink semantics dataflow of a small problem (b).

represented by the graph in fig. 3, which represents the semantics dataflow of a simple map-reduce program. The user program translation into the semantics dataflow can be subject to further optimizations. For instance, two or more non-intensive kernels can be mapped onto the same actor to reduce resource usage. In Google TensorFlow, semantics dataflow maps directly to the graph described at the user-level API, as shown in Fig. 2(a). Since some edges may have different meanings (i.e. explicit control dependencies), the dataflow at this level gives a specialized view on all the different semantics of nodes and edges.

Apache Spark and Flink directly map the user application to the semantics dataflow level: nodes represent operations (either transformations or actions, in the Spark nomenclature) that take as input collections, represented by RDDs in Spark and streams in Flink. Fig. 4(a) shows the semantics dataflow related to the WordCount application, having as operations (in order): 1. read from text file; 2. a `flatMap` operator splitting the file into words; 3. a `map` operator that maps each word into a key-value pair $\langle w, 1\rangle$; 4. a `reduceByKey` operator that counts occurrences of each word in the input file. It is interesting to note that the DAG is grouped into stages (namely, stages 0 and 1). Those stages divide "map" phases from "reduce" phases, where a "reduce" phase implies a data exchange possibly among all processing elements, called *shuffle*. This aspect will be covered in more detail later. Apache Flink calls its DAG semantics dataflow the JobGraph (or *condensed view* in the Flink nomenclature): a representation of the application consisting of operators (JobVertex) and intermediate results (IntermediateDataSet, representing data dependencies among

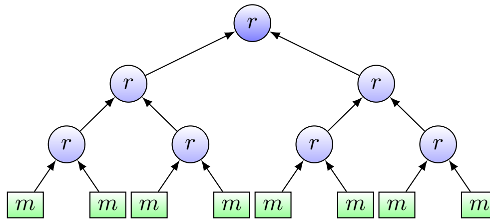


Figure 5: MapReduce execution dataflow with maximum level of parallelism reached by eight *map* instances.

operators). A small example of a JobGraph is shown in Fig. 4(b).

5.2 Data Granularity

We note that, since all the frameworks provide at this level a DAG representing the application semantics, they share a similar expressiveness. A key difference, though, is related to what tokens moved along the edges. In a batch program, the data model represents a collection of data flowing across actors and the output of each operator/actor is a new collection or a single value of any type. In this scenario, actors are "one-shot": they are no longer available to the system after they produce the output since operators consume only one input token. This differs from a stream processing application, where actors are "kept alive" to consume new input tokens. Data model for stream processing applications provided by those frameworks differ with respect to the granularity, which has consequences on operations implemented in each actor: frameworks can be divided into fine vs. coarse grain data processing, that is, single item or micro-batch. The varying granularity means that actors operate on collections of items (Spark and windows in Flink) or on single items (Storm, Flink). A coarser granularity limits the possibility to implement rich algorithms based on window operations and limits the real-time property at the basis of every streaming application, since time is spent in collecting a certain amount of data to create a single token. Apache Storm works at single item granularity in the base form, while exploiting microbatches when using the Trident extension. Trident is a high level abstraction on top of Storm, that can be used as an alternative to the regular API, providing a functional level such as the one described in the first layer of our layered model in Fig. 1. Trident processes streams as a series of micro-batches which are called transactions.

6 Parallel Execution Dataflow

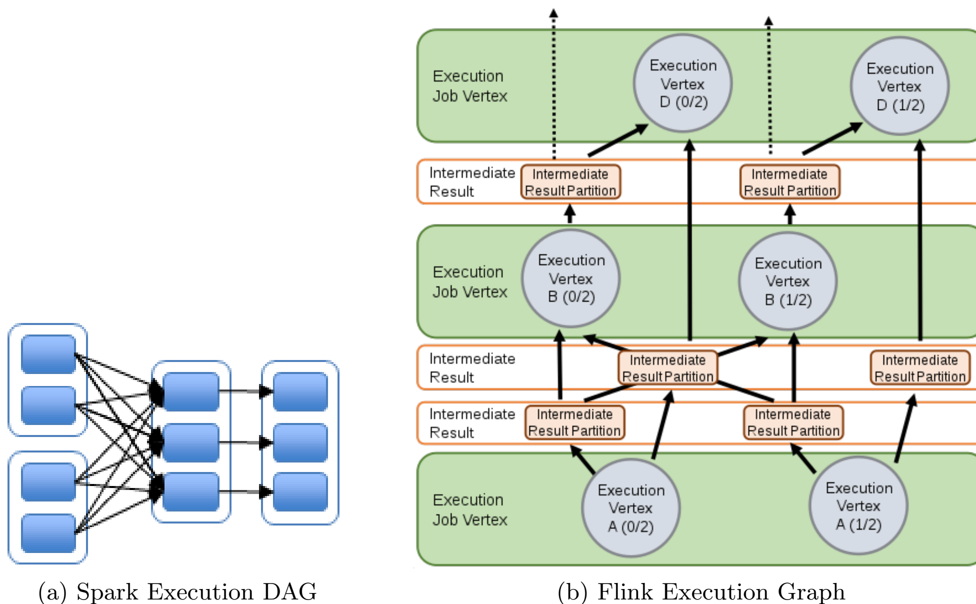
This level represents the effective instantiation of the semantics dataflow, in which some actors are replicated to increase the level of parallelism with respect to the upper level, where all parallelism came from independent actors. For some collective operations (e.g., `map` defined in Sec. 4.1), each replica works over a *partition* of the input data. This schema is generally referred as *embarrassingly parallel* processing: since input data of replicas are disjoint, there are no dependencies among replicas.

Referring to the MapReduce example presented in Fig. 3 with its semantics dataflow, its parallel execution dataflow can now be represented as in Fig. 5. In this example, the dataset A is divided in 8 independent partitions and the map function m is executed by

8 actor replicas; the reduce phase is then executed in parallel by actors enabled by the incoming tokens (namely, the results) from its "producer" actors. As in the previous section, we describe how the considered frameworks instantiate the dataflow and what are the consequences it brings to the runtime.

Apache Spark identifies its parallel execution dataflow by a DAG such as the one shown in Fig. 6(a), which is the input of the DAG Scheduler entity. This graph illustrates two main aspects: first, the fact that many parallel instances of actors are created for each function and, second, the operators grouping. This grouping identifies the so called *Stages* that are executed in parallel if and only if there is no dependency among them. The stage grouping brings another strong consequence, derived from the implementation of the Spark runtime: each stage that depends on one or more previous stages has to wait for their completion before starting execution. The depicted behavior is analogous to the one encountered in the Bulk Synchronous Parallelism paradigm (BSP) [17]. In a BSP algorithm, as well as in a Spark application, a computation proceeds in a series of global *supersteps* consisting in: 1) Concurrent computation, in which every actor executes its business code on its own partition of data; 2) Communication, where processes exchange data between themselves if necessary (the so called *shuffle* phase); 3) Barrier synchronization, where processes wait until all other processes have reached the same barrier. That is, the firing rule of the actor is a *from-all* activation rule.

This has an important consequence on the Spark Streaming side. Since the runtime is the same for both batch and stream processing, we can infer that in a stream application, each micro-batch has to go over the whole DAG before the next one can be triggered.



(a) Spark Execution DAG (b) Flink Execution Graph

Figure 6: Parallel execution dataflow of a simple Map/Reduce application in Spark and Flink.

Apache Flink transforms the JobGraph (Fig. 4(b)) into another graph called the ExecutionGraph [8] (Fig. 6(b)), which is a parallel version of the JobGraph, where the JobVertex is an abstract vertex containing a certain number of ExecutionVertexes, one per parallel subtask. Each operator with a parallelism of, for example, 100 will have one JobVertex and 100 ExecutionVertices. A key difference with respect to the Spark execution graph is that a dependency does not represent a barrier among execution vertexes (or stages in Spark); instead, there is effective pipelining so tasks can be executed concurrently. This is a natural implementation for stream processing, but in this case, since the runtime is the same, it applies to batch processing applications as well. Conversely, iterative processing is implemented according to the BSP approach: one evaluation of the step function on all parallel instances forms a *superstep*, which is also the granularity of synchronization; all parallel tasks of an iteration need to complete the superstep before the next one is initiated, thus behaving like a *barrier* between iterations.

Apache Storm creates an environment for the execution dataflow similar to the Spark execution graph. Each actor is replicated to increase the parallelism and each group of replicas is identified by the name of the Bolt/Spout of the semantics dataflow they originally belong to. Each of these collective actors represents data parallel tasks without dependencies. Since Storm is a stream processing framework, pipeline parallelism is implemented. Hence, while one operator is processing a tuple, an upstream operator can process the next tuple concurrently, increasing both data parallelism within each actors group and task parallelism among groups.

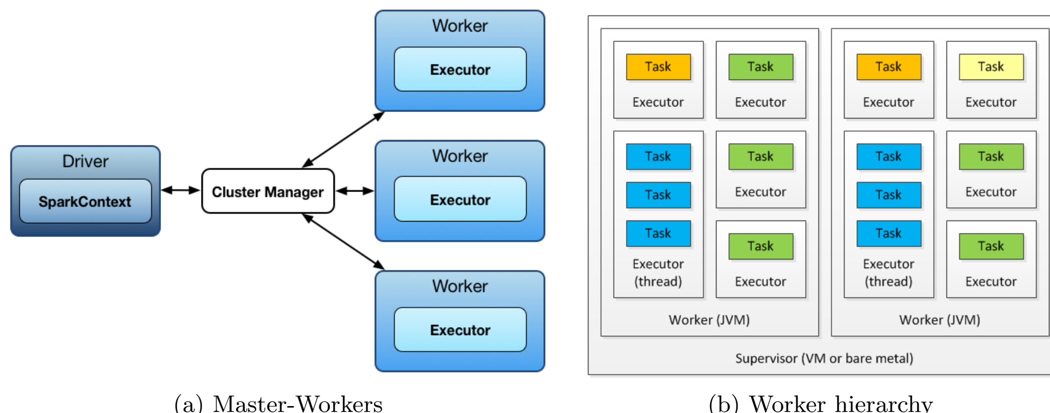
Google TensorFlow replicates vertexes, which denote tensor operators according to the data parallel approach. This dataflow model also corresponds to the parallel execution dataflow, since each operator is data parallel and works on data elements—here, multidimensional arrays (tensors). Hence we can state that each node is a data-parallel actor operating on intra-task independent input elements. Moreover, iterative nodes are implemented with a notion of tags similar to the MIT Tagged-Token dataflow machine [6], where the iteration state is identified by a tag and independent iterations are executed in parallel. It is worthwhile to remark TensorFlow differs from Flink in the execution of iterative nodes: in TensorFlow an input can enter a loop iteration whenever it becomes available, while Flink poses a barrier after each iteration.

Summarizing, in this section, we showed how the analyzed frameworks are using the very same model to represent the application that will be executed, namely, a dataflow directed acyclic graph.

7 Dataflow Process Network

This layer shows how the program is effectively executed. In all frameworks but TensorFlow, the resulting process dataflow follows the Master-Workers pattern, where actors on previous layers are transformed into tasks. In TensorFlow, actors are effectively mapped to threads and possibly distributed on different nodes.

Fig. 7(a) shows a representation of the Spark Master-Workers runtime. We will use this structure also to examine Storm and Flink, since the pattern is the same: all that changes is



(a) Master-Workers (b) Worker hierarchy

Figure 7: Master-Workers structure of the Spark runtime (a) and Worker hierarchy example in Storm (b).

how tasks are distributed among workers and how the inter/intra-communication between actors is managed.

The Master has total control over program execution, job scheduling, communications, failure management, resource allocations, etc. The master also relies on a cluster manager, an external service for acquiring resources on the cluster (like Mesos, YARN or Zookeper). The master is the one that knows the semantic dataflow representing the current application, while workers are completely agnostic about the whole dataflow: they only obtain tasks to execute, tasks that represent nodes of the semantic dataflow the master is executing. This is possible since programs are executed lazily, that is, it is only when the execution is effectively launched that the semantic dataflow is evaluated and eventually optimized to obtain the best execution plan (Flink). In Storm and Flink, the data distribution is managed in a decentralized manner, that is, it is delegated to each executor, since they use pipelined data transfers and forward tuples as soon as they are produced. In Spark streaming, the master is the one responsible for data distribution: it discretizes the stream into micro-batches that are buffered into workers' memory. The master generally keeps track of distributed tasks, decides when to schedule the next tasks, reacts to finished/failed tasks, keeps track of the semantic dataflow progress, and orchestrates collective communications and data exchange among workers. This last job is crucial when executing a shuffle operation, that needs data exchange among executors. Whereas workers do not have any information about other workers, to exchange data they have to request informations from the master and, moreover, specify they are ready to send/receive data.

Workers are nodes executing the operators code, namely, a worker node is a process in the cluster. Within a worker, a certain number of parallel executors is instantiated, that execute tasks related to the given application. Workers have no information about the dataflow at any level since they are scheduled by the master. Despite this, the different frameworks use different nomenclatures: in Spark, Storm and Flink cluster nodes are decomposed into *Workers*, *Executors* and *Tasks*. A Worker is a process in a node of the cluster, i.e., a Spark worker instance. A node may host multiple Worker instances. An Executor is a thread that

is spawned in a Worker process and it executes Tasks, which are the actual kernel of a node of the dataflow. Fig. 7(b) illustrates this structure in Storm, an example that would also be valid for Spark and Flink.

Moving to TensorFlow, the cardinality of the semantic dataflow is preserved, as each operator node is instantiated into one node, and the placement is decided using a placement algorithm cost model. The dataflow is distributed on cluster nodes and each node/Worker may host one or more dataflow nodes/Tasks, that internally implement data parallelism with a pool of threads/Executors working on Tensors. Communications among nodes is done using the send-receive paradigm, allowing workers to manage their own data movement or receiving data without involving the master node, thus decentralizing the logic and the execution of the application (see also Fig. 2(b)).

8 Limitations of the Dataflow model

Reasoning about programs by means of the Dataflow model is attractive since it makes the program semantics independent from the underlying execution model. In particular, it abstracts away any form of parallelism due to its pure functional nature. The most relevant consequence, as discussed in many theoretical works about Kahn Process Network and similar models—such as Dataflow—is the fact that all computations are deterministic.

Conversely, many parallel runtime systems exploit nondeterministic behaviors in order to provide efficient implementations. For example, consider the Master-Workers pattern we discussed in Sec. 7. A naive implementation of the Master node distributes tasks to N Workers according to a round-robin policy—task i goes to worker $i\,(mod\ N)$—which leads to a deterministic process. An alternative policy, generally referred as *on-demand*, distributes tasks by considering the load level of each worker, for example, to implement a form of load balancing. The resulting processes are clearly nondeterministic, since the mapping from tasks to workers depends on the relative service times.

Non-determinism can be encountered at all levels of our layered model in Fig. 1. For example, nodes of Storm's topologies consume tokens from incoming streams according to a from-any policy—process an item from any non-empty input channel—thus no assumption can be made about the order in which stream items are processed. More generally, the semantics of stateful streaming programs depend on the order in which stream items are processed, which is not specified by the semantics of the semantic Dataflow nodes in Sec. 5. As a consequence, this prevents from reasoning in purely Dataflow—thus functional—terms about programs in which actor nodes include arbitrary code in some imperative language (e.g., shared variables).

9 From Skeletons to Big Data, a Historical Perspective

The need to exploit parallel computing at a high enough level of abstraction certainly cannot be attributed to the advent (or the hype) of Big Data. In the parallel computing and software engineering communities, this need has been advocated years before by way of algorithmic skeletons [9] and design patterns [12], respectively. Conceptually, the tools we discussed through the paper exploit *Data Parallelism, Stream Parallelism*, or both.

In the framework of Big Data oriented tools, data parallelism is often realised by way of the MapReduce paradigm [11], for instance, as implemented by way of the Apache Hadoop

framework. Certainly pragmatically revolutionary, from the parallelism exploitation viewpoint, the MapReduce paradigm is a specialisation of the *Map* and *Reduce* paradigm composition with a powerful reduce operator oriented to data analytics workloads (i.e., `groupByKey`). The ability to efficiently support lists (tensors, actually) tranformations under weakly ordered execution models has been proved by Gorlatch's seminal work [14], which definitely influenced the design of MapReduce. Also, Map, Reduce and other data parallel skeletons have been introduced and developed in many experimental parallel programming frameworks; most of them described in the survey[3] from González-Vélez and Leyton [13].

Stream parallelism is an equally important parallelism exploitation pattern, from the first high-level approaches to parallel computing, such as the P3L language [7], to the recent frameworks, such as FastFlow [5]. In the Big Data context, currently, stream parallelism is interpreted in a quite primitive way considering only basic management of data streams, such as micro-batching, which turns stream parallelism into data parallelism, and processing of independent data streams. More advanced usage of streams can be found in [10].

Patterns/skeletons share many of the principles underlying the high-level frameworks we considered in the previous sections. Data Parallel patterns express computations in which the same kernel function is applied to all items of a data collection, which include Map and Reduce. They can be viewed as higher-order functions and can be placed at the very top of our layered model from Fig. 1, since they expose a declarative data processing model (Sec. 4.1).

Stream Parallel patterns express computations in which data streams flow through a network of processing units. This model, enriched with Control-Parallel patterns such as `If` and `While`, allows to express programs in terms of arbitrary graphs, where vertexes are processing units and edges are network links. In this setting, Stream Parallel patterns represent pre-built, nestable graphs, therefore they expose a topological data processing model (Sec. 4.2).

Parallel patterns can be composed to express arbitrarily complex data processing programs. We already informally showed they subsume both declarative and topological data-processing models. Moreover, TensorFlow can be regarded as a particular case of the two-tier composition paradigm advocated, e.g., in [4], in which Data Parallel patterns can be embedded only into Stream Parallel patterns, thus resulting into a hybrid declarative-topological model.

10 Conclusion

In this paper, we showed how the Dataflow model can be used to describe Big Data analytics tools, from the lowest level—process execution model—to the highest one—semantic dataflow. The Dataflow model is expressive enough to represent computations in term of batch, micro-batch and stream processing. With this abstraction, we showed that Big Data analytics tools have similar expressiveness at all levels and we proceeded with the description of a layered model capturing different levels of Big Data applications, from the program semantics to the execution model. We also provided a survey of some well-known tools—Apache Spark, Flink, Storm and TensorFlow—by analyzing their semantics and mapping them to the proposed Dataflow-based layered model. With this work, we aim at giving users

[3]Now a bit outdated but still among the most comprehensive surveys in this area.

a general framework to understand the model underlying all the analyzed tools. Finally, we also showed how the skeleton-based model provides an alternative but similar abstraction, briefly describing some of most common parallel patterns that easily describe data parallel patterns that are used for Big Data analytics. As future work, we plan to implement a model of Big Data analytics tools based on algorithmic skeletons, on top of the FastFlow library [5].

¡¡¡¡¡¡¡ Updated upstream

=======

¿¿¿¿¿¿¿ Stashed changes

References

[1] Apache Flink website. `https://flink.apache.org/`

[2] Abadi, M., et al.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems (2015). URL `http://tensorflow.org/`

[3] Aldinucci, M., Danelutto, M., Anardu, L., Torquati, M., Kilpatrick, P.: Parallel patterns + macro data flow for multi-core programming. In: Proc. of Intl. Euromicro PDP 2012: Parallel Distributed and network-based Processing, pp. 27–36. IEEE, Garching, Germany (2012)

[4] Aldinucci, M., Danelutto, M., Drocco, M., Kilpatrick, P., Pezzi, G.P., Torquati, M.: The Loop-of-Stencil-Reduce paradigm. In: Proc. of Intl. Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms (RePara), pp. 172–177. IEEE, Helsinki, Finland (2015)

[5] Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: Accelerating code on multi-cores with FastFlow. In: Proc. of 17th Intl. Euro-Par 2011 Parallel Processing, *LNCS*, vol. 6853, pp. 170–181. Springer, Bordeaux, France (2011)

[6] Arvind, K., Nikhil, R.S.: Executing a program on the MIT tagged-token dataflow architecture. IEEE Trans. Comput. **39**(3), 300–318 (1990)

[7] Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneschi, M.: P3L: a structured high level programming language and its structured support. Concurrency Practice and Experience **7**(3), 225–255 (1995)

[8] Carbone, P., Fóra, G., Ewen, S., Haridi, S., Tzoumas, K.: Lightweight asynchronous snapshots for distributed dataflows. CoRR **abs/1506.08603** (2015)

[9] Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge, MA, USA (1991)

[10] De Matteis, T., Mencagli, G.: Parallel patterns for window-based stateful operators on data streams: an algorithmic skeleton approach. International Journal of Parallel Programming pp. 1–20 (2016)

[11] Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008)

[12] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley (1995)

[13] González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. Softw. Pract. Exper. **40**(12), 1135–1160 (2010)

[14] Gorlatch, S.: Systematic efficient parallelization of scan and other list homomorphisms. In: Proc. of the 2nd Intl. Euro-Par Conference on Parallel Processing-Volume II, Euro-Par '96, pp. 401–408. Springer-Verlag, London, UK (1996)

[15] Lee, E.A., Parks, T.M.: Dataflow process networks. Proc. of the IEEE **83**(5), 773–801 (1995)

[16] Nasir, M.A.U., Morales, G.D.F., García-Soriano, D., Kourtellis, N., Serafini, M.: The power of both choices: Practical load balancing for distributed stream processing engines. CoRR **abs/1504.00788** (2015)

[17] Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)

[18] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In: Proc. of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12. USENIX, Berkeley, CA, USA (2012)

[19] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: Fault-tolerant streaming computation at scale. In: Proc. of the 24th ACM Symposium on Operating Systems Principles, SOSP, pp. 423–438. ACM, New York, NY, USA (2013)