A Comparison of Big Data Frameworks on a Layered Dataflow Model

Parallel Processing Letters

A COMPARISON OF BIG DATA FRAMEWORKS ON A LAYERED DATAFLOW MODEL

CLAUDIA MISALE, MAURIZIO DROCCO, MARCO ALDINUCCI*
Dept. of Computer Science, University of Torino, Italy

GUY TREMBLAY†
Dépt. d'informatique, Université du Québec à Montréal, Canada

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

In the world of Big Data analytics, there is a series of tools aiming at simplifying programming applications to be executed on clusters. Although each tool claims 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 model we propose 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 proposed model is (at least) as general as existing batch and streaming frameworks (e.g., Spark, Flink, Storm), thus making it easier to understand 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.

Keywords: data processing, streaming, dataflow, skeletons, functional programming, semantics

1. Outline

With the increasing number of Big Data analytics tools, we witness a continuous fight among implementors/vendors in demonstrating how their tools are better than others in terms of performances or expressiveness. In this hype, for a user approaching Big Data analytics (even an educated computer scientist), it might be difficult to have a clear picture of the programming model underneath these tools and the expressiveness they provide to solve some user defined problem. With this in mind,

*{*misale, drocco, aldinuc*}*@di.unito.it*

†*tremblay.guy@uqam.ca*

we wanted to understand the features those tools provide to the user in terms of API and how they were related to parallel computing paradigms.

To provide some order in the world of Big Data processing, in this paper we categorize some models and tools to identify their programming model's common features. We identified the *Dataflow model* [12] as the common model that better describes all levels of abstraction, from the user-level API to the execution model. This model represents applications as a directed graph of actors. In its "modern" reissue (aka. macro-data flow [2]), it naturally models independent (thus parallelizable) kernels starting from a graph of true data dependencies, where a kernel's 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 arising 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 arise when processing one or more streams in one node of the graph, a well-known problem in parallel and distributed computing. Finally, the paper also focuses 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 from a Dataflow perspective: Spark [15], Storm [13], Flink [9], and TensorFlow [1]. We focus only on those tools since they are among the most famous and used ones nowadays. As far as we know, no previous attempt has been made to compare different Big Data processing tools, at multiple levels of abstraction, under a common formalism.

The paper proceeds as follows. Section 2 describes the Dataflow model and how it can be exploited at three different abstraction levels. Section 3 focuses on user-level API of the tools. The various levels of our layered model are discussed in Sections 4, 5 and 6. Then, Section 7 discusses some limitations of the dataflow model in capturing all the tools' features. Finally, Section 8 concludes the paper and describes some future work.

2. The Dataflow Layered Model

By analyzing some well-known tools—Spark, Storm, Flink, and TensorFlow—we identified a common structure underlying all of them, based on the Dataflow model. In Section 2.1 we review the Dataflow model of computation, as presented by Lee and Parks [12]. In Section 2.2, we outline an architecture that can describe all these models at different levels of abstraction (see Fig. 1) from the (top) user-level

API to the (bottom-level) actual network of processes. In particular, we show how the Dataflow model is general enough to subsume many different levels only by changing the semantics of actors and channels.

2.1. *The Dataflow Model*

Dataflow Process Networks are a special case of *Kahn Process Networks*, a model of computation that describes a program as a set of concurrent processes communicating with each other via FIFO channels, where reads are blocking and writes are non-blocking [11]. In a Dataflow process network, a set of *firing rules* is associated with each process, called *actor*. Processing then 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 functional actors are stateless—and the output tokens are functions of the input tokens. The model can also be extended to allow stateful actors.

A Dataflow network can be executed mainly using two approaches, namely *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 and different processes communicate 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 unit being scheduled 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 arriving from one or more streams. The function mapping input to output tokens is called the *kernel* of an actor. The Dataflow Process Network model also seamlessly comprehends the Macro Dataflow parallel execution model, in which each process executes arbitrary code. Conversely, an actor's code in a classical Dataflow *architecture* model is typically a single machine instruction.

A *firing rule* defines when an actor can fire. Each rule defines what tokens have to be available for the actor to fire. In the basic model, one token from each input channel must be available in order to enable one firing of the actor (i.e., from-all input policy). 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, to express arbitrary stream consuming policies (e.g., gathering from any channel: cf. Section 7).

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 channels, 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 internal state. Stateful actors is an extension that allows side effects over *local* (i.e., internal to each actor) states. As shown by Lee and Sparks [12], stateful actors can be emulated in the stateless 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 and by defining appropriate firing rules.

2.2. *The Dataflow Stack*

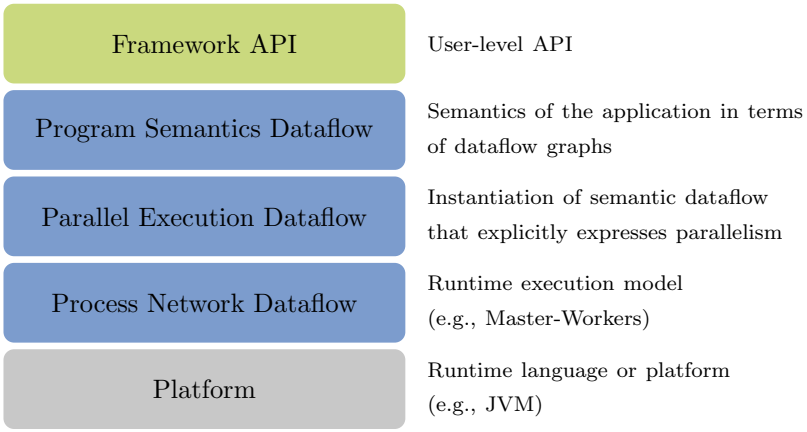


Fig. 1. Layered model representing the levels of abstractions provided by the frameworks that were analyzed.

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 which is beyond the scope of our paper. On top is the *Framework API* level, that describes the user API on top of the Dataflow graph, which will be detailed in Section 3. 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 program components (i.e., actors). This aspect is covered in Section 4.

- *Parallel Execution Dataflow*: This level, covered in Section 5, represents an 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.
- *Process Network Dataflow*: This level, covered in Section 6, 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. In TensorFlow, processing elements are effectively mapped to threads and possibly distributed over multiple nodes of a cluster.

3. The Frameworks' User APIs

Data-processing applications are generally divided into *batch* vs. *stream* processing. Batch programs process one or more *finite* datasets to produce a resulting finite output dataset, whereas stream programs process possibly unbounded sequences of data, called *streams*, doing so in an incremental manner. Operations over streams may also have to respect a total data ordering—for instance, to represent time ordering.

Orthogonally, we divide the frameworks' user APIs into two categories: *declarative* and *topological*. Spark, Flink, and TensorFlow belong to the first category—they provide batch or stream processing in the form of operators over collections or streams—whereas Storm belong to the second one—it provides an API explicitly based on building graphs.

3.1. *Declarative Data Processing*

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

Declarative batch processing applications are expressed as methods on objects representing collections (Spark and Flink) or as functions on values (*tensors*, in TensorFlow): these are algebras on finite datasets, whose data can be ordered (as in tensors) or not (as in Spark/Flink multisets). APIs with such operations are exposing a functional-like style. Here are three examples of operations with their

(multiset-based) semantics:[a]

$$\texttt{groupByKey}(a) = \{(k, \{v : (k, v) \in a\})\} \quad (1)$$

$$\texttt{join}(a, b) = \{(k, (v_a, v_b)) : (k, v_a) \in a \wedge (k, v_b) \in b\} \quad (2)$$

$$\texttt{map}\langle f \rangle(a) = \{f(v) : v \in a\} \quad (3)$$

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.

Declarative 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 used, for example, to express the position of each stream item with respect to a global order—a global timestamp—or to partition streams into substreams. For instance, this allows expressing relational algebra operators and data grouping. In a stream processing scenario, we also have to consider two important aspects: state and windowing; those are discussed in Section 3.3.

Apache Spark implements batch programming with a set of operators, called *transformations*, that are uniformly applied to whole datasets called *Resilient Distributed Datasets* (RDD) [15], which are immutable multisets. For stream processing, Spark implements an extension through the Spark Streaming module, providing a high-level abstraction called *discretized stream* or *DStream* [16]. Such streams represent results in continuous sequences of RDDs of the same type, called *micro-batch*. Operations over DStreams are "forwarded" to each RDD in the DStream, thus the semantics of operations over streams is defined in terms of batch processing according to the simple translation $\texttt{op}(a) = [\texttt{op}(a_1), \texttt{op}(a_2), \ldots]$, where $[\cdot]$ refers to a possibly unbounded ordered sequence, $a = [a_1, a_2, \ldots]$ is a DStream, and each item a_i is a micro-batch of type RDD.

Listing in Fig. 2 shows code for the simple Word Count example in Spark—the "Hello World!" example for Big Data. A collection (RDD) of `words` is first created by scanning a text file and splitting each line into its constituent words. Each word `s` is then paired (`Tuple2`) with 1, to indicate one occurrence of that word, generating the `pairs` RDD. All the 1s for a given word are then combined together, and reduced using addition, to obtain RDD `counts`, whose result is then saved as a text file.

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 composed (i.e, pipelined) by calling operators on DataStream objects. Flink also provides the DataSet type for batch applications, that identifies a single immutable multiset—a stream of one element. A Flink program, either for stream or batch

[a]Here, $\{\cdot\}$ denotes *multisets* rather than sets.

```
  sc.textFile("hdfs://...");
JavaRDD<String> words =
  textFile.flatMap(new FlatMapFunction<String, String>() {
      public Iterable<String> call(String s) {
        return Arrays.asList(s.split(" "));
      }
    });
JavaPairRDD<String, Integer> pairs =
  words.mapToPair(new PairFunction<String, String, Integer>() {
      public Tuple2<String, Integer> call(String s) {
        return new Tuple2<String, Integer>(s, 1);
      }
    });
JavaPairRDD<String, Integer> counts =
  pairs.reduceByKey(new Function2<Integer, Integer, Integer>() {
      public Integer call(Integer a, Integer b) {
        return a + b;
      }
    });
counts.saveAsTextFile("hdfs://...");
```

Fig. 2. Word Count example in Spark.

```
  public static void main(String[] args) throws Exception {
    final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
    DataSet<String> text = env.fromElements("Text...");
    DataSet<Tuple2<String, Integer>> wordCounts =
      text
      .flatMap(new LineSplitter())
      .groupBy(0)
      .sum(1);

    wordCounts.print();
  }

  public static class LineSplitter
              implements FlatMapFunction<String, Tuple2<String, Integer>> {
    @Override
    public void flatMap(String line, Collector<Tuple2<String, Integer>> out) {
      for (String word : line.split(" ")) {
        out.collect(new Tuple2<String, Integer>(word, 1));
      }
    }
  }
}
```

Fig. 3. Word Count example in Flink.

processing, is a term from an algebra of operators over DataStreams or DataSets, respectively. Stateful stream operators and iterative batch processing are discussed in Section 3.3.

Listing in Fig. 3 shows Flink's code for the Word Count example. The `text DataSet` is the sequence of lines from some text file. The resulting `wordCounts DataSet`, again consisting of words paired with their number of occurrences (`Tuple2<String, Integer>`), is obtained by splitting each line into words paired with 1s (`flatMap` with `LineSplitter`), then grouped by word (0^{th} component of pair) and summed over the various 1s (1^{st} commponent of pair).

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 composition of operators processing tensors. A TensorFlow application is built as a functional-style expression, where each subexpression can be given an explicit name. The TensorFlow programming model includes control flow operations and, notably, synchronization primitives (e.g., *MutexAquire*/*MutexRelease* for critical sections). This latter observation implies TensorFlow exposes the underlying (parallel) execution model to the user which has to program the eventual coordination of operators concurring over some global state. Because of space limitation, we do not provide a TensorFlow Word Count example.

3.2. *Topological Data Processing*

Topological programs are expressed as graphs, built by explicitly connecting processing nodes and specifying the code executed by nodes.

Apache Storm is a framework that only targets stream processing. Storm's programming model is based on three key notions: *Spouts*, *Bolts*, and *Topologies*. A Spout is a source of a stream, which 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. A Topology is the composition of Spouts and Bolts resulting in a network. Storm uses *tuples* as its data model, i.e., named lists of values of arbitrary type. Hence, Bolts are parametrized with per-tuple kernel code. Each time a tuple is available from some input stream, the kernel code gets activated to process that input tuple. Bolts and Spouts are locally stateful, as we discuss in Section 3.3, while no global consistent state is supported. Yet, globally stateful computations can be implemented since the kernel code of Spouts and Bolts is arbitrary. However, eventual global state management would be the sole responsibility of the user, who has to be aware of the underlying execution model in order ensure program coordination among Spouts and Bolts. It is also possible to define cyclic graphs by way of feedback channels connecting Bolts.

While Storm targets single-tuple granularity in its base interface, the Trident API is an abstraction that provides declarative stream processing on top of Storm. Namely, Trident processes streams as a series of micro-batches belonging to a stream considered as a single object.

Listing in Fig. 4 shows Storm's code for the Word Count example. A key element is the `WordCount` *bolt* `execute` method. Each call to `execute` receives a `Tuple`, a `word`. The *bolt* keeps track of the number of occurrrences of each word using the `counts Map` (lines 19, 28), and emits that `word` paired with its current `count` (29)—thus generating a stream of incremental number of occurrences. The spout (random sentences) and bolts (sentence splitter, word counting) are created and connected in the `main` method.

```
public static class SplitSentence extends ShellBolt implements IRichBolt {
  public SplitSentence() {
    super("python", "splitsentence.py");
  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("word"));
  }

  @Override
  public Map<String, Object> getComponentConfiguration() {
    return null;
  }
}

public static class WordCount extends BaseBasicBolt {
  Map<String, Integer> counts = new HashMap<String, Integer>();

  @Override
  public void execute(Tuple tuple, BasicOutputCollector collector) {
    String word = tuple.getString(0);
    Integer count = counts.get(word);
    if (count == null)
      count = 0;
    count++;
    counts.put(word, count);
    collector.emit(new Values(word, count));
  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("word", "count"));
  }
}

public static void main(String[] args) throws Exception {
  TopologyBuilder builder = new TopologyBuilder();
  builder.setSpout("spout", new RandomSentenceSpout(), 5);
  builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("spout");
  builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new Fields("word"))
      ;
  Config conf = new Config();
  conf.setDebug(true);
  conf.setNumWorkers(3);
  StormSubmitter.submitTopology(args[0], conf, builder.createTopology());
}
}
```

Fig. 4. Word Count example in Storm.

3.3. *State, Windowing and Iterative Computations*

Frameworks providing *stateful* stream 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. 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.

Windowing is another concept provided by many stream processing frameworks. A *window* is informally defined as an ordered subset of items extracted from the stream. The most common form of windowing is referred as a *sliding window*, characterized by its size (how many elements fall within the window) and sliding policy (how items enter and exit from the window). Spark provides the simplest abstraction for defining windows, since they are just micro-batches over the DStream abstraction, where only the window size and sliding policy can be specified. Storm and Flink allow more arbitrary kinds of grouping, producing windows of Tuples and WindowedStreams, respectively. Note that this does not break the declarative or topological nature of the considered frameworks, since it only changes the type of the processed data. Note also that windowing can be expressed in terms of stateful processing, by considering window-typed state.

Finally, we consider another common concept in batch processing, namely *iterative* processing. In Flink, iterations are expressed as the composition of arbitrary DataSet values by iterative operators, resulting in a so-called *IterativeDataSet*. Component DataSets represent for example *step functions*—executed in each iteration—or termination condition—evaluated to decide if iteration has to be terminated. Spark's iteration model is radically simpler, since no specific construct is provided to implement iterative processing. Instead, an RDD (endowed with transformations) can be embedded into a plain sequential loop. Finally, TensorFlow allows expressing conditionals and loops by means of specific control flow operators such as *For*, similarly to Flink.

4. Program Semantics Dataflow

The Program Semantics Dataflow level of our layered model provides a representation of the program in terms of the Dataflow model. Such a model describes the application using operators and data dependencies among them, thus creating a topological view common to all frameworks. 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.

4.1. *Semantic Dataflow Graphs*

A semantic Dataflow graph is a pair $G = \langle V, E \rangle$ where actors V represent operators, channels E represent data dependencies among operators and tokens represent data to be processed. For instance, consider a map function m followed by a reduce function r on a collection A and its result b, represented as the functional composition $b = r(m(A))$. This is represented by the graph in Fig. 5, which represents the semantic dataflow of a simple map-reduce program. Note that the user program

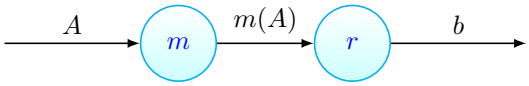


Fig. 5. Functional Map and Reduce dataflow expressing data dependencies.

translation into the semantic dataflow can be subject to further optimization. For instance, two or more non-intensive kernels can be mapped onto the same actor to reduce resource usage.

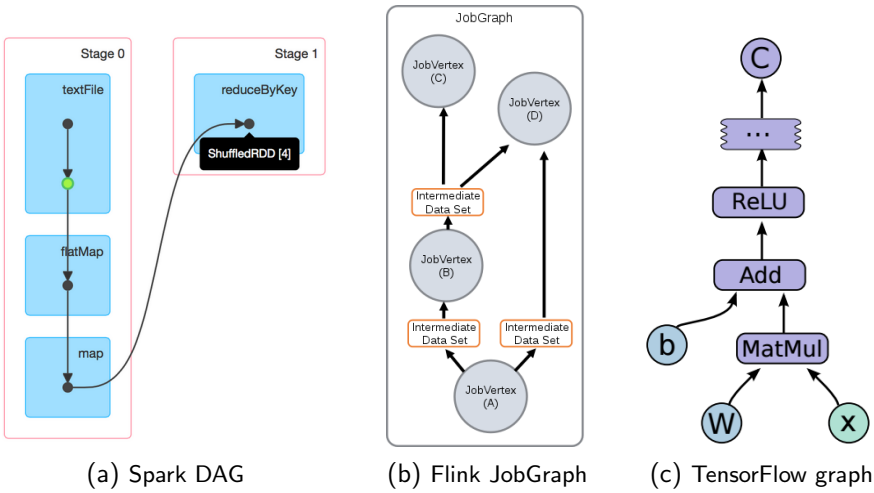


Fig. 6. Spark DAG of the WordCount application (a). A Flink JobGraph (b). A TensorFlow application graph, adapted from [1] (c).

Notably, the Dataflow representation we propose is adopted by the considered frameworks as a pictorial representation of applications. Fig. 6(a) shows the semantic dataflow—called application DAG in Spark—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 $(w, 1)$; 4. a `reduceByKey` operator that counts occurrences of each word in the input file. The DAG is grouped into *stages* (namely, Stages 0 and 1), which divide *map* and *reduce* phases. This distinction is related to the underlying parallel execution model and will be covered in Section 5. Flink also provides a semantic representation—called JobGraph or *condensed view*— of the application, consisting of operators (JobVertex) and intermediate results (IntermediateDataSet, representing data dependencies among operators). Fig. 6(b) presents a small example of a JobGraph. Finally, Fig. 6(c) is a TensorFlow example (adapted from [1]). A node represents a tensor operation, which can be also a data generation node (e.g., W, b, x). Each node has firing rules that depend on the kind of incoming tokens. For example, *control dependencies* edges can carry synchronization tokens: the target

node of such edges cannot execute until all appropriate synchronization signals have been received.

4.2. *Tokens and Actors Semantics*

Although the frameworks provide a similar semantic expressiveness, some differences are visible regarding the meaning of tokens flowing across channels and how many times actors are activated.

When mapping a Spark program, tokens represent RDDs and DStreams for batch and stream processing respectively. Actors are operators—either transformations or actions in Spark nomenclature—that transform data or return values (in-memory collection or files). Actors are activated only once in both batch and stream processing, since each collection (either RDD or DStreams) is represented by a single token. For Flink, the approach is similar: actors are activated only once in all scenarios except in iterative algorithms—see Sect. 4.3. Tokens represent DataSets and DataStreams that identify whole datasets and streams respectively. For TensorFlow, the same mapping holds: operators are mapped to actors that take as input single tokens representing Tensors (multi-dimensional arrays). Actors are activated once except for iterative computations, as in Flink. Storm is different since a token represents a single stream item (Tuple). Consequently, actors, representing (macro) dataflow operators, are activated each time a new token is available.

From the discussion above, we can note that Storm's actors follow a *from-any* policy for consuming input tokens, while the other frameworks follow a *from-all* policy as in the basic Dataflow model. In all the considered frameworks, output tokens are broadcast onto all channels going out of a node.

4.3. *Semantics of State, Windowing and Iterations*

In Section 3.3, we introduced stateful, windowing and iterative processing as convenient tools provided by the considered frameworks.

From a Dataflow perspective, stateful actors represent an extension to the basic model—as sketched in Section 2.1—only in case of global state. In particular, globally-stateful processing breaks the functional nature of the basic Dataflow model, inhibiting for instance to reason in pure functional terms about program semantics (cf. Section 7). Conversely, locally-stateful processing can be emulated in terms of the pure Dataflow model. We remark that at the semantic level, capturing stateful processing within declarative models requires no modifications to the proposed Dataflow model, since this aspect is embedded into the semantics of each operation.

For instance, consider the semantics of a generic `mapWithState` functional. This functional is parametrized by the binary kernel $f : T \times S \to U \times S$, that takes as input an item to be processed ($a_i \in T$) in addition to the state ($s_i \in S$), and then produces an output item in addition to a new state. Let s_0 be the initial state and a_i be the i^{th} item from an arbitrary ordering of collection a. The semantics of the

generic invocation of the kernel, with value s_i for the state, can then be defined as follows, for $i \geq 1$:

$$y_i = f(a_{i-1}, s_{i-1})$$
$$s_i = \pi_2(y_i)$$

The semantics of the stateful functional, where Π_1 is the left projection over a whole collection, is then the following:

$$\texttt{mapWithState}\langle f \rangle(a) = \Pi_1 \left(\bigcup y_i \right)$$

The above semantics is clearly non-deterministic since it depends on the ordering choice. A similar formulation also holds for partitioned states, but in that case the binary kernel takes as input a subset of the state (i.e., the portion bound with the respective key); equivalently, it produces an update for the same subset of the state.

Moreover, windowing is not a proper extension since windows can be stored within each actor's local state [8]. However, the considered frameworks treat windowing as a primitive concept. This can be easily mapped to the Dataflow domain by just considering tokens of proper types.

Finally, iterations can be modeled by inserting loops in semantic dataflows. In this case, each actor involved in an iteration is activated each time a new token is available and the termination condition is not met. This implementation of iterative computations is similar to the hierarchical actors of Lee & Parks [12], used to encapsulate subgraphs modeling iterative algorithms.

5. Parallel Execution Dataflow

The Parallel Execution Dataflow level represents parallel implementations of semantic dataflows. As in the previous section, we start by introducing the approach and then we describe how the various frameworks instantiate it and what are the consequences this brings to the runtime.

The most straightforward source of parallelism comes directly from the Dataflow model, namely, independent actors can run in parallel. Furthermore, some actors can be replicated to increase parallelism by making replicas work over a *partition* of the input data—that is, by exploiting full *data parallelism*. This is the case, for instance, of the `map` operator described in Section 3.1. Both the above schemas are referred as *embarrassingly parallel* processing, since there are no dependencies among actors. Note that introducing data parallelism requires partitioning input tokens into sub-tokens, distributing those to the various worker replicas, and then aggregating the resulting sub-tokens into an appropriate result token—much like `scatter/gather` operations in message passing programs. Finally, in case of dependent actors that are activated multiple times, parallelism can still be exploited by letting tokens "flow" as soon as each activation is completed. This well-known schema is referred as *stream/pipeline* parallelism.

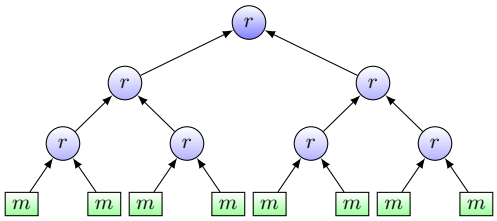
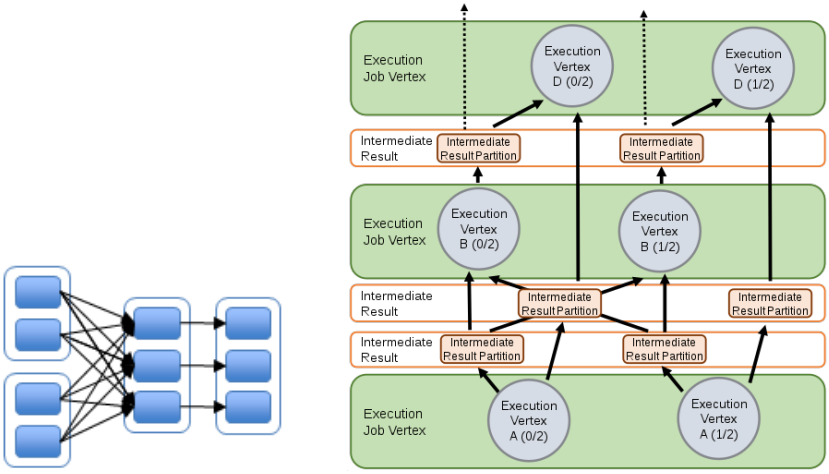


Fig. 7. MapReduce execution dataflow with maximum level of parallelism reached by eight *map* instances.

Figure 7 shows a parallel execution dataflow for the MapReduce semantic dataflow from 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 their "producer" actors.



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

Fig. 8. Parallel execution dataflow of a simple Map/Reduce application in Spark and Flink.

Spark identifies its parallel execution dataflow by a DAG such as the one shown in Fig. 8(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 actors are grouped into *Stages* that are executed in parallel if and only if there is no dependency among them. Stages can be considered as the hierarchical actors in [12]. Grouping actors in stages brings another consequence, derived from the Spark runtime implementation: each stage that depends on some previous stages has to wait for their completion before execution. The depicted behavior is analogous to the one encountered in the Bulk

Synchronous Parallelism paradigm (BSP) [14]. 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 each actor executes its business code on its own partition of data; 2) Communication, where actors exchange data between themselves if necessary (the *shuffle* phase); 3) Barrier synchronization, where actors wait until all other actors have reached the same barrier.

Flink transforms a JobGraph (e.g., Fig. 6(b)) into an ExecutionGraph [6] (e.g., Fig. 8(b)), in which the JobVertex (a hierarchical actor) is an abstract vertex containing ExecutionVertexes (actors), one per parallel sub-task. A key difference compared to the Spark execution graph is that a dependency does not represent a barrier among actors or hierarchical actors: instead, there is effective tokens pipelining, and thus actors can be fired 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* (again a hierarchical actor), 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.

TensorFlow replicates actors implementing certain operators (e.g., tensor multiplication) on tensors (input tokens). Hence, each actor is a data-parallel actor operating on intra-task independent input elements—here, multi-dimensional arrays (tensors). Moreover, iterative actors/hierarchical actors (in case of cycles on a subgraph) are implemented with tags similar to the MIT Tagged-Token dataflow machine [4], where the iteration state is identified by a tag and independent iterations are executed in parallel. It is interesting to note that TensorFlow differs from Flink in the execution of iterative actors: in TensorFlow an input can enter a loop iteration whenever it becomes available, while Flink imposes a barrier after each iteration.

Storm creates an environment for the execution dataflow similar to the other frameworks. Each actor is replicated to increase the inter-actor parallelism and each group of replicas is identified by the name of the Bolt/Spout of the semantics dataflow they originally belong to, thus instantiating a hierarchical actor. Each of these actors (actors group) represents data parallel tasks without dependencies. Since Storm is a stream processing framework, pipeline parallelism is exploited. Hence, while an actor is processing a token (tuple), an upstream actor can process the next token concurrently, increasing both data parallelism within each actors group and task parallelism among groups.

Summarizing, in Sections 4 and 5, we showed how the considered frameworks can be compared through the lens of the very same model from both a semantic and a parallel implementation perspective. The comparison is summarized in Table 1 and Table 2 for batch and streaming processing, respectively.

Table 1. Batch processing.

	Spark	**Flink**	**TensorFlow**
Graph specification	Implicit, OO-style chaining of transformations	Implicit, OO-style chaining of transformations	Implicit, Prefix operator with arguments
DAG	Join operation	Join operation	N-ary operators and/or results
Tokens	RDD	DataSet	Tensor
Nodes	Transformations from RDD to RDD	Transformations from DataSet to DataSet	Transformations from Tensor to Tensor
Parallelism	Data parallelism in transformations + Inter-actor, task parallelism, limited by per-stage BSP	Data parallelism in transformations + Inter-actor task parallelism	Data parallelism in transformations + Inter-actor task parallelism + Loop parallelism
Iteration	Using *repetitive sequential executions* of the graph	Using `iterate` & `iterateDelta`	Using control flow constructs

Table 2. Stream processing.

	Spark	**Flink**	**Storm**
Graph specification	Implicit, OO-style chaining of transformations	Implicit, OO-style chaining of transformations	Explicit, Connections between *bolts*
DAG	Join operation	Join operation	Multiple incoming/outgoing connections
Tokens	DStream	DataStream	Tuple (fine-grain)
Nodes	Transformations from DStream to DStream	Transformations from DataStream to DataStream	Stateful with "arbitrary" emission of output tuples
Parallelism	Analogous to Spark Batch parallelism	Analogous to Flink Batch parallelism + Stream parallelism between stream items	Data parallelism between different bolt instances + Stream parallelism between stream items by bolts

6. Dataflow Process Network

Th Process Network layer shows how the program is effectively executed, following the process and scheduling-based categorization described earlier (Sect. 2.1).

6.1. *Scheduling-based Execution*

In Spark, Flink and Storm, the resulting process network dataflow follows the Master-Workers pattern, where actors from previous layers are transformed into

tasks. Fig. 9(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 similar for them: they differ only in how tasks are distributed among workers and how the inter/intra-communication between actors is managed.

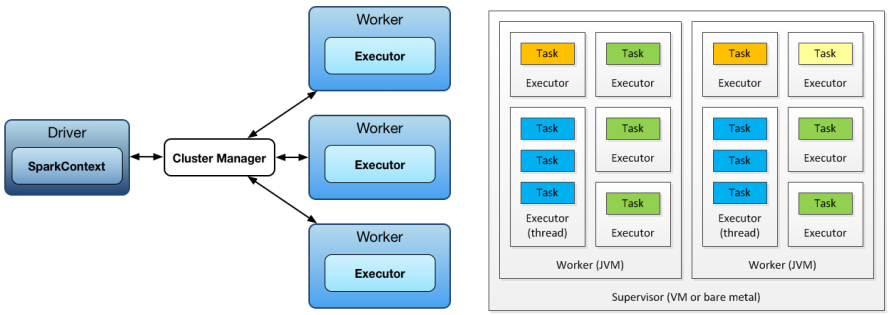


Fig. 9. Master-Workers structure of the Spark runtime (a) and Worker hierarchy example in Storm (b).

The Master has total control over program execution, job scheduling, communications, failure management, resource allocations, etc. The master is the entity that knows the semantic dataflow representing the current application, while workers are completely agnostic about the whole dataflow: they only obtain tasks to execute, that represent actors of the execution dataflow the master is running. It is only when the execution is effectively launched that the semantic dataflow is built and eventually optimized to obtain the best execution plan (Flink). With this postponed evaluation, the master creates what we called the parallel execution dataflow to be executed. In Storm and Flink, the data distribution is managed in a decentralized manner, i.e., it is delegated to each executor, since they use pipelined data transfers and forward tokens as soon as they are produced. In Spark streaming, the master is 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 vs. failed tasks, keeps track of the semantic dataflow progress, and orchestrates collective communications and data exchange among workers. This last aspect is crucial when executing *shuffle operations*, which entail data exchanges among executors. Whereas workers do not have any information about others, to exchange data they have to request information to the master and, moreover, specify they are ready to send/receive data.

Workers are nodes executing the actor logic, 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, e.g., 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 an actor of the dataflow. Fig. 9(b) illustrates this structure in Storm, an example that would also be valid for Spark and Flink.

6.2. *Process-based Execution*

In TensorFlow, actors are effectively mapped to threads and possibly distributed on different nodes. The cardinality of the semantic dataflow is preserved, as each actor node is instantiated into one node, and the allocation is decided using a placement algorithm based on a cost model. The dataflow is distributed on cluster nodes and each node/Worker may host one or more dataflow actors/Tasks, that internally implement data parallelism with a pool of threads/Executors working on Tensors. Communication among actors is done using the send/receive paradigm, allowing workers to manage their own data movement or to receive data without involving the master node, thus decentralizing the logic and the execution of the application.

7. Limitations of the Dataflow Model

Reasoning about programs using 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 to provide efficient implementations. For example, consider the Master-Workers pattern discussed in Section 6. A naive implementation of the Master node distributes tasks to N Workers according to a round-robin policy—task i goes to worker $i \,(\text{MOD } N)$—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, actors in Storm's topologies consume tokens from incoming streams according to a from-any policy—process a token from any non-empty input channel—thus no assumption can be made about the order in which stream tokens are processed. More generally, the semantics of stateful streaming programs depends on the order in which stream items are processed, which is not specified by the semantics of

the semantic dataflow actors in Section 4. As a consequence, this prevents from reasoning in purely Dataflow—i.e., functional—terms about programs in which actor nodes include arbitrary code in some imperative language (e.g., shared variables).

8. 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 terms 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 an overview of some well-known tools—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 a general model to understand the levels underlying all the analyzed tools.

The need to exploit parallel computing at a high enough level of abstraction certainly predates the advent (or the "hype") of Big Data processing. In the parallel computing and software engineering communities, this need has been advocated years before by way of algorithmic skeletons [7] and design patterns [10], which share many of the principles underlying the high-level frameworks considered in previous sections. Conceptually, the tools we discussed through the paper exploit *Data Parallelism, Stream Parallelism*, or both.

Data Parallel patterns express computations in which the same kernel function is applied to all items of a data collection, which include for instance 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 (Section 3.1).

Stream Parallel patterns express computations in which data streams flow through a network of processing units. It is another key parallelism exploitation pattern, from the first high-level approaches to parallel computing, such as the P3L language [5], to more recent frameworks, such as FastFlow [3]. This model, enriched with Control-Parallel patterns such as `If` and `While`, allows to express programs through 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 (Section 3.2).

As future work, we plan to implement a model of Big Data analytics tools based on algorithmic skeletons, on top of the FastFlow library [3], exploiting both forms of parallelism.

Acknowledgements This work was partly supported by the EU-funded project TOREADOR (contract no. H2020-688797), the EU-funded project Rephrase (contract no. H2020-644235), and the 2015–2016 IBM Ph.D. Scholarship program. We

gratefully acknowledge Prof. Domenico Talia for his comments on the early version of the manuscript.

References

[1] M. Abadi, A. Agarwal, P. Barham, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[2] M. Aldinucci, M. Danelutto, L. Anardu, M. Torquati, and P. Kilpatrick. Parallel patterns + macro data flow for multi-core programming. In *Proc. of Intl. Euromicro PDP 2012*, pages 27–36, Garching, Germany, Feb. 2012. IEEE.

[3] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. Accelerating code on multi-cores with FastFlow. In *Proc. of 17th Intl. Euro-Par 2011 Parallel Processing*, volume 6853 of *LNCS*, pages 170–181, Bordeaux, France, Aug. 2011. Springer.

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

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

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

[7] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1991.

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

[9] Apache Flink website. `https://flink.apache.org/`, 2016 (last accessed).

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[11] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*, pages 471–475, 1974.

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

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

[14] L. G. Valiant. A bridging model for parallel computation. *CACM*, 33(8):103–111, Aug. 1990.

[15] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. 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, Berkeley, CA, USA, 2012. USENIX.

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